Red Hat JBoss Fuse 6.1

Apache CXF Development Guide

Develop applications with Apache CXF Web services

Last Updated: 2017-10-12

Red Hat JBoss Fuse 6.1 Apache CXF Development Guide

Develop applications with Apache CXF Web services

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

REVISIT -

Table of Contents

PART I. WRITING WSDL CONTRACTS

Abstract

This part describes how to define a Web service interface using WSDL.

CHAPTER 1. INTRODUCING WSDL CONTRACTS

Abstract

WSDL documents define services using Web Service Description Language and a number of possible extensions. The documents have a logical part and a concrete part. The abstract part of the contract defines the service in terms of implementation neutral data types and messages. The concrete part of the document defines how an endpoint implementing a service will interact with the outside world.

The recommended approach to design services is to define your services in WSDL and XML Schema before writing any code. When hand-editing WSDL documents you must make sure that the document is valid, as well as correct. To do this you must have some familiarity with WSDL. You can find the standard on the W3C web site, www.w3.org.

1.1. STRUCTURE OF A WSDL DOCUMENT

A WSDL document is, at its simplest, a collection of elements contained within a root **`definition`** element. These elements describe a service and how an endpoint implementing that service is accessed.

A WSDL document has two distinct parts:

- A logical part that defines the service in implementation neutral terms
- A concrete part that defines how an endpoint implementing the service is exposed on a network

The logical part

The logical part of a WSDL document contains the **`types`**, the **`message`**, and the **`portType`** elements. It describes the service's interface and the messages exchanged by the service. Within the **`types`** element, XML Schema is used to define the structure of the data that makes up the messages. A number of **`message`** elements are used to define the structure of the messages used by the service. The **`portType`** element contains one or more **`operation`** elements that define the messages sent by the operations exposed by the service.

The concrete part

The concrete part of a WSDL document contains the **`binding`** and the **`service`** elements. It describes how an endpoint that implements the service connects to the outside world. The **`binding`** elements describe how the data units described by the **`message`** elements are mapped into a concrete, on-the-wire data format, such as SOAP. The **`service`** elements contain one or more **`port`** elements which define the endpoints implementing the service.

1.2. WSDL ELEMENTS

A WSDL document is made up of the following elements:

- **`definitions`** – The root element of a WSDL document. The attributes of this element specify the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced in the WSDL document.
- **`types`** – The XML Schema definitions for the data units that form the building blocks of the

messages used by a service. For information about defining data types see Chapter 2, *Defining Logical Data Units*.

- **`message`** — The description of the messages exchanged during invocation of a services operations. These elements define the arguments of the operations making up your service. For information on defining messages see Chapter 3, *Defining Logical Messages Used by a Service*.
- **`portType`** — A collection of **`operation`** elements describing the logical interface of a service. For information about defining port types see Chapter 4, *Defining Your Logical Interfaces*.
- **`operation`** — The description of an action performed by a service. Operations are defined by the messages passed between two endpoints when the operation is invoked. For information on defining operations see the section called "Operations".
- **`binding`** — The concrete data format specification for an endpoint. A **`binding`** element defines how the abstract messages are mapped into the concrete data format used by an endpoint. This element is where specifics such as parameter order and return values are specified.
- **`service`** — A collection of related **`port`** elements. These elements are repositories for organizing endpoint definitions.
- **`port`** — The endpoint defined by a binding and a physical address. These elements bring all of the abstract definitions together, combined with the definition of transport details, and they define the physical endpoint on which a service is exposed.

1.3. DESIGNING A CONTRACT

To design a WSDL contract for your services you must perform the following steps:

1. Define the data types used by your services.
2. Define the messages used by your services.
3. Define the interfaces for your services.
4. Define the bindings between the messages used by each interface and the concrete representation of the data on the wire.
5. Define the transport details for each of the services.

CHAPTER 2. DEFINING LOGICAL DATA UNITS

Abstract

When describing a service in a WSDL contract complex data types are defined as logical units using XML Schema.

When defining a service, the first thing you must consider is how the data used as parameters for the exposed operations is going to be represented. Unlike applications that are written in a programming language that uses fixed data structures, services must define their data in logical units that can be consumed by any number of applications. This involves two steps:

1. Breaking the data into logical units that can be mapped into the data types used by the physical implementations of the service
2. Combining the logical units into messages that are passed between endpoints to carry out the operations

This chapter discusses the first step. Chapter 3, *Defining Logical Messages Used by a Service*discusses the second step.

2.1. MAPPING DATA INTO LOGICAL DATA UNITS

The interfaces used to implement a service define the data representing operation parameters as XML documents. If you are defining an interface for a service that is already implemented, you must translate the data types of the implemented operations into discreet XML elements that can be assembled into messages. If you are starting from scratch, you must determine the building blocks from which your messages are built, so that they make sense from an implementation standpoint.

Available type systems for defining service data units

According to the WSDL specification, you can use any type system you choose to define data types in a WSDL contract. However, the W3C specification states that XML Schema is the preferred canonical type system for a WSDL document. Therefore, XML Schema is the intrinsic type system in Apache CXF.

XML Schema as a type system

XML Schema is used to define how an XML document is structured. This is done by defining the elements that make up the document. These elements can use native XML Schema types, like xsd:int, or they can use types that are defined by the user. User defined types are either built up using combinations of XML elements or they are defined by restricting existing types. By combining type definitions and element definitions you can create intricate XML documents that can contain complex data.

When used in WSDL XML Schema defines the structure of the XML document that holds the data used to interact with a service. When defining the data units used by your service, you can define them as types that specify the structure of the message parts. You can also define your data units as elements that make up the message parts.

Considerations for creating your data units

You might consider simply creating logical data units that map directly to the types you envision using when implementing the service. While this approach works, and closely follows the model of building RPC-style applications, it is not necessarily ideal for building a piece of a service-oriented architecture.

The Web Services Interoperability Organization's WS-I basic profile provides a number of guidelines for defining data units and can be accessed at http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES. In addition, the W3C also provides the following guidelines for using XML Schema to represent data types in WSDL documents:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.

2.2. ADDING DATA UNITS TO A CONTRACT

Depending on how you choose to create your WSDL contract, creating new data definitions requires varying amounts of knowledge. The Apache CXF GUI tools provide a number of aids for describing data types using XML Schema. Other XML editors offer different levels of assistance. Regardless of the editor you choose, it is a good idea to have some knowledge about what the resulting contract should look like.

Procedure

Defining the data used in a WSDL contract involves the following steps:

1. Determine all the data units used in the interface described by the contract.
2. Create a **types** element in your contract.
3. Create a **schema** element, shown in Example 2.1, "Schema entry for a WSDL contract" , as a child of the **type** element.

   The **targetNamespace** attribute specifies the namespace under which new data types are defined. The remaining entries should not be changed.

   Example 2.1. Schema entry for a WSDL contract

   ```
   <schema targetNamespace="http://schemas.iona.com/bank.idl"
           xmlns="http://www.w3.org/2001/XMLSchema"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   ```

4. For each complex type that is a collection of elements, define the data type using a **complexType** element. See Section 2.4.1, "Defining data structures".
5. For each array, define the data type using a **complexType** element. See Section 2.4.2, "Defining arrays".
6. For each complex type that is derived from a simple type, define the data type using a **simpleType** element. See Section 2.4.4, "Defining types by restriction".
7. For each enumerated type, define the data type using a **simpleType** element. See Section 2.4.5, "Defining enumerated types".
8. For each element, define it using an **element** element. See Section 2.5, "Defining elements".

2.3. XML SCHEMA SIMPLE TYPES

If a message part is going to be of a simple type it is not necessary to create a type definition for it. However, the complex types used by the interfaces defined in the contract are defined using simple types.

Entering simple types

XML Schema simple types are mainly placed in the **`element`** elements used in the types section of your contract. They are also used in the **`base`** attribute of **`restriction`** elements and **`extension`** elements.

Simple types are always entered using the **`xsd`** prefix. For example, to specify that an element is of type int, you would enter xsd:int in its **`type`** attribute as shown in Example 2.2, “Defining an element with a simple type”.

Example 2.2. Defining an element with a simple type

```
<element name="simpleInt" type="xsd:int" />
```

Supported XSD simple types

Apache CXF supports the following XML Schema simple types:

- xsd:string
- xsd:normalizedString
- xsd:int
- xsd:unsignedInt
- xsd:long
- xsd:unsignedLong
- xsd:short
- xsd:unsignedShort
- xsd:float
- xsd:double
- xsd:boolean
- xsd:byte
- xsd:unsignedByte
- xsd:integer
- xsd:positiveInteger

- xsd:negativeInteger
- xsd:nonPositiveInteger
- xsd:nonNegativeInteger
- xsd:decimal
- xsd:dateTime
- xsd:time
- xsd:date
- xsd:QName
- xsd:base64Binary
- xsd:hexBinary
- xsd:ID
- xsd:token
- xsd:language
- xsd:Name
- xsd:NCName
- xsd:NMTOKEN
- xsd:anySimpleType
- xsd:anyURI
- xsd:gYear
- xsd:gMonth
- xsd:gDay
- xsd:gYearMonth
- xsd:gMonthDay

2.4. DEFINING COMPLEX DATA TYPES

XML Schema provides a flexible and powerful mechanism for building complex data structures from its simple data types. You can create data structures by creating a sequence of elements and attributes. You can also extend your defined types to create even more complex types.

In addition to building complex data structures, you can also describe specialized types such as enumerated types, data types that have a specific range of values, or data types that need to follow certain patterns by either extending or restricting the primitive types.

2.4.1. Defining data structures

In XML Schema, data units that are a collection of data fields are defined using **complexType** elements. Specifying a complex type requires three pieces of information:

1. The name of the defined type is specified in the **name** attribute of the **complexType** element.
2. The first child element of the **complexType** describes the behavior of the structure's fields when it is put on the wire. See the section called "Complex type varieties".
3. Each of the fields of the defined structure are defined in **element** elements that are grandchildren of the **complexType** element. See the section called "Defining the parts of a structure".

For example, the structure shown in Example 2.3, "Simple Structure" is be defined in XML Schema as a complex type with two elements.

Example 2.3. Simple Structure

```
struct personalInfo
{
  string name;
  int age;
};
```

Example 2.4, "A complex type" shows one possible XML Schema mapping for the structure shown in Example 2.3, "Simple Structure".

Example 2.4. A complex type

```
<complexType name="personalInfo">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
  </sequence>
</complexType>
```

Complex type varieties

XML Schema has three ways of describing how the fields of a complex type are organized when represented as an XML document and passed on the wire. The first child element of the **complexType** element determines which variety of complex type is being used. Table 2.1, "Complex type descriptor elements" shows the elements used to define complex type behavior.

Table 2.1. Complex type descriptor elements

Element	Complex Type Behavior
sequence	All the complex type's fields must be present and they must be in the exact order they are specified in the type definition.

Element	Complex Type Behavior
all	All of the complex type's fields must be present but they can be in any order.
choice	Only one of the elements in the structure can be placed in the message.

If a **sequence** element, an **all** element, or a **choice** is not specified, then a **sequence** is assumed. For example, the structure defined in Example 2.4, "A complex type" generates a message containing two elements: **name** and **age**.

If the structure is defined using a **choice** element, as shown in Example 2.5, "Simple complex choice type", it generates a message with either a **name** element or an **age** element.

Example 2.5. Simple complex choice type

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </choice>
</complexType>
```

Defining the parts of a structure

You define the data fields that make up a structure using **element** elements. Every **complexType** element should contain at least one **element** element. Each **element** element in the **complexType** element represents a field in the defined data structure.

To fully describe a field in a data structure, **element** elements have two required attributes:

- The **name** attribute specifies the name of the data field and it must be unique within the defined complex type.
- The **type** attribute specifies the type of the data stored in the field. The type can be either one of the XML Schema simple types, or any named complex type that is defined in the contract.

In addition to **name** and **type**, **element** elements have two other commonly used optional attributes: **minOcurrs** and **maxOccurs**. These attributes place bounds on the number of times the field occurs in the structure. By default, each field occurs only once in a complex type. Using these attributes, you can change how many times a field must or can appear in a structure. For example, you can define a field, **previousJobs**, that must occur at least three times, and no more than seven times, as shown in Example 2.6, "Simple complex type with occurrence constraints" .

Example 2.6. Simple complex type with occurrence constraints

```
<complexType name="personalInfo>
  <all>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
```

```
    <element name="previousJobs" type="xsd:string:
             minOccurs="3" maxOccurs="7"/>
  </all>
</complexType>
```

You can also use the **minOccurs** to make the **age** field optional by setting the **minOccurs** to zero as shown in Example 2.7, "Simple complex type with minOccurs set to zero" . In this case **age** can be omitted and the data will still be valid.

Example 2.7. Simple complex type with minOccurs set to zero

```
<complexType name="personalInfo>
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int" minOccurs="0"/>
  </choice>
</complexType>
```

Defining attributes

In XML documents attributes are contained in the element's tag. For example, in the **complexType** element **name** is an attribute. They are specified using the **attribute** element. It comes after the **all**, **sequence**, or **choice** element and are a direct child of the **complexType** element. Example 2.8, "Complex type with an attribute" shows a complex type with an attribute.

Example 2.8. Complex type with an attribute

```
<complexType name="personalInfo>
  <all>
    <element name="name" type="xsd:string"/>
    <element name="previousJobs" type="xsd:string"
             minOccurs="3" maxOccurs="7"/>
  </all>
  <attribute name="age" type="xsd:int" use="optional" />
</complexType>
```

The **attribute** element has three attributes:

- **name** – A required attribute that specifies the string identifying the attribute.
- **type** – Specifies the type of the data stored in the field. The type can be one of the XML Schema simple types.
- **use** – Specifies if the attribute is required or optional. Valid values are **required** or **optional**.

If you specify that the attribute is optional you can add the optional attribute **default**. The **default** attribute allows you to specify a default value for the attribute.

2.4.2. Defining arrays

Apache CXF supports two methods for defining arrays in a contract. The first is define a complex type with a single element whose **maxOccurs** attribute has a value greater than one. The second is to use SOAP arrays. SOAP arrays provide added functionality such as the ability to easily define multi-dimensional arrays and to transmit sparsely populated arrays.

Complex type arrays

Complex type arrays are a special case of a sequence complex type. You simply define a complex type with a single element and specify a value for the **maxOccurs** attribute. For example, to define an array of twenty floating point numbers you use a complex type similar to the one shown in Example 2.9, "Complex type array".

Example 2.9. Complex type array

```
<complexType name="personalInfo">
  <element name="averages" type="xsd:float" maxOccurs="20"/>
</complexType>
```

You can also specify a value for the **minOccurs** attribute.

SOAP arrays

SOAP arrays are defined by deriving from the SOAP-ENC:Array base type using the **wsdl:arrayType** element. The syntax for this is shown in Example 2.10, "Syntax for a SOAP array derived using wsdl:arrayType".

Example 2.10. Syntax for a SOAP array derived using wsdl:arrayType

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

Using this syntax, *TypeName* specifies the name of the newly-defined array type. *ElementType* specifies the type of the elements in the array. *ArrayBounds* specifies the number of dimensions in the array. To specify a single dimension array use **[]**; to specify a two-dimensional array use either **[][]** or **[,]**.

For example, the SOAP Array, SOAPStrings, shown in Example 2.11, "Definition of a SOAP array", defines a one-dimensional array of strings. The **wsdl:arrayType** attribute specifies the type of the array elements, xsd:string, and the number of dimensions, with **[]** implying one dimension.

Example 2.11. Definition of a SOAP array

```
<complexType name="SOAPStrings">
  <complexContent>
```

```
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="xsd:string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in Example 2.12, “Syntax for a SOAP array derived using an element” .

Example 2.12. Syntax for a SOAP array derived using an element

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <sequence>
        <element name="ElementName" type="ElementType"
                 maxOccurs="unbounded"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

When using this syntax, the element's **maxOccurs** attribute must always be set to **unbounded**.

2.4.3. Defining types by extension

Like most major coding languages, XML Schema allows you to create data types that inherit some of their elements from other data types. This is called defining a type by extension. For example, you could create a new type called **alienInfo**, that extends the **personalInfo** structure defined in Example 2.4, “A complex type” by adding a new element called **planet**.

Types defined by extension have four parts:

1. The name of the type is defined by the **name** attribute of the **complexType** element.
2. The **complexContent** element specifies that the new type will have more than one element.

NOTE

If you are only adding new attributes to the complex type, you can use a **simpleContent** element.

3. The type from which the new type is derived, called the *base* type, is specified in the **base** attribute of the **extension** element.
4. The new type’s elements and attributes are defined in the **extension** element, the same as they are for a regular complex type.

For example, **alienInfo** is defined as shown in Example 2.13, “Type defined by extension”.

Example 2.13. Type defined by extension

```
<complexType name="alienInfo">
  <complexContent>
    <extension base="personalInfo">
      <sequence>
        <element name="planet" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

2.4.4. Defining types by restriction

XML Schema allows you to create new types by restricting the possible values of an XML Schema simple type. For example, you can define a simple type, **SSN**, which is a string of exactly nine characters. New types defined by restricting simple types are defined using a **simpleType** element.

The definition of a type by restriction requires three things:

1. The name of the new type is specified by the **name** attribute of the **simpleType** element.
2. The simple type from which the new type is derived, called the *base type*, is specified in the **restriction** element. See the section called "Specifying the base type".
3. The rules, called *facets*, defining the restrictions placed on the base type are defined as children of the **restriction** element. See the section called "Defining the restrictions".

Specifying the base type

The base type is the type that is being restricted to define the new type. It is specified using a **restriction** element. The **restriction** element is the only child of a **simpleType** element and has one attribute, **base**, that specifies the base type. The base type can be any of the XML Schema simple types.

For example, to define a new type by restricting the values of an xsd:int you use a definition like the one shown in Example 2.14, "Using int as the base type".

Example 2.14. Using int as the base type

```
<simpleType name="restrictedInt">
  <restriction base="xsd:int">
    ...
  </restriction>
</simpleType>
```

Defining the restrictions

The rules defining the restrictions placed on the base type are called *facets*. Facets are elements with one attribute, **value**, that defines how the facet is enforced. The available facets and their valid **value** settings depend on the base type. For example, xsd:string supports six facets, including:

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whitespace**
- **enumeration**

Each facet element is a child of the **restriction** element.

Example

Example 2.15, "SSN simple type description" shows an example of a simple type, **SSN**, which represents a social security number. The resulting type is a string of the form **xxx-xx-xxxx**. <SSN>032-43-9876<SSN> is a valid value for an element of this type, but <SSN>032439876</SSN> is not.

Example 2.15. SSN simple type description

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}"/>
  </restriction>
</simpleType>
```

2.4.5. Defining enumerated types

Enumerated types in XML Schema are a special case of definition by restriction. They are described by using the **enumeration** facet which is supported by all XML Schema primitive types. As with enumerated types in most modern programming languages, a variable of this type can only have one of the specified values.

Defining an enumeration in XML Schema

The syntax for defining an enumeration is shown in Example 2.16, "Syntax for an enumeration".

Example 2.16. Syntax for an enumeration

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value"/>
    <enumeration value="Case2Value"/>
    ...
    <enumeration value="CaseNValue"/>
  </restriction>
</simpleType>
```

EnumName specifies the name of the enumeration type. *EnumType* specifies the type of the case

values. *CaseNValue*, where *N* is any number one or greater, specifies the value for each specific case of the enumeration. An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.

Example

For example, an XML document with an element defined by the enumeration **widgetSize**, shown in Example 2.17, "widgetSize enumeration", would be valid if it contained <widgetSize>big</widgetSize>, but it would not be valid if it contained <widgetSize>big,mungo</widgetSize>.

Example 2.17. widgetSize enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
  </restriction>
</simpleType>
```

2.5. DEFINING ELEMENTS

Elements in XML Schema represent an instance of an element in an XML document generated from the schema. The most basic element consists of a single **element** element. Like the **element** element used to define the members of a complex type, they have three attributes:

- **name** — A required attribute that specifies the name of the element as it appears in an XML document.
- **type** — Specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. This attribute can be omitted if the type has an in-line definition.
- **nillable** — Specifies whether an element can be omitted from a document entirely. If **nillable** is set to **true**, the element can be omitted from any document generated using the schema.

An element can also have an *in-line* type definition. In-line types are specified using either a **complexType** element or a **simpleType** element. Once you specify if the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data. In-line type definitions are discouraged because they are not reusable.

CHAPTER 3. DEFINING LOGICAL MESSAGES USED BY A SERVICE

Abstract

A service is defined by the messages exchanged when its operations are invoked. In a WSDL contract these messages are defined using **`message`** element. The messages are made up of one or more parts that are defined using **`part`** elements.

A service's operations are defined by specifying the logical messages that are exchanged when an operation is invoked. These logical messages define the data that is passed over a network as an XML document. They contain all of the parameters that are a part of a method invocation.

Logical messages are defined using the **`message`** element in your contracts. Each logical message consists of one or more parts, defined in **`part`** elements.

TIP

While your messages can list each parameter as a separate part, the recommended practice is to use only a single part that encapsulates the data needed for the operation.

MESSAGES AND PARAMETER LISTS

Each operation exposed by a service can have only one input message and one output message. The input message defines all of the information the service receives when the operation is invoked. The output message defines all of the data that the service returns when the operation is completed. Fault messages define the data that the service returns when an error occurs.

In addition, each operation can have any number of fault messages. The fault messages define the data that is returned when the service encounters an error. These messages usually have only one part that provides enough information for the consumer to understand the error.

MESSAGE DESIGN FOR INTEGRATING WITH LEGACY SYSTEMS

If you are defining an existing application as a service, you must ensure that each parameter used by the method implementing the operation is represented in a message. You must also ensure that the return value is included in the operation's output message.

One approach to defining your messages is RPC style. When using RPC style, you define the messages using one part for each parameter in the method's parameter list. Each message part is based on a type defined in the **`types`** element of the contract. Your input message contains one part for each input parameter in the method. Your output message contains one part for each output parameter, plus a part to represent the return value, if needed. If a parameter is both an input and an output parameter, it is listed as a part for both the input message and the output message.

RPC style message definition is useful when service enabling legacy systems that use transports such as Tibco or CORBA. These systems are designed around procedures and methods. As such, they are easiest to model using messages that resemble the parameter lists for the operation being invoked. RPC style also makes a cleaner mapping between the service and the application it is exposing.

MESSAGE DESIGN FOR SOAP SERVICES

While RPC style is useful for modeling existing systems, the service's community strongly favors the wrapped document style. In wrapped document style, each message has a single part. The message's part references a wrapper element defined in the **types** element of the contract. The wrapper element has the following characteristics:

- It is a complex type containing a sequence of elements. For more information see Section 2.4, "Defining complex data types".
- If it is a wrapper for an input message:
  - It has one element for each of the method's input parameters.
  - Its name is the same as the name of the operation with which it is associated.
- If it is a wrapper for an output message:
  - It has one element for each of the method's output parameters and one element for each of the method's inout parameters.
  - Its first element represents the method's return parameter.
  - Its name would be generated by appending **Response** to the name of the operation with which the wrapper is associated.

MESSAGE NAMING

Each message in a contract must have a unique name within its namespace. It is recommended that you use the following naming conventions:

- Messages should only be used by a single operation.
- Input message names are formed by appending **Request** to the name of the operation.
- Output message names are formed by appending **Response** to the name of the operation.
- Fault message names should represent the reason for the fault.

MESSAGE PARTS

Message parts are the formal data units of the logical message. Each part is defined using a **part** element, and is identified by a **name** attribute and either a **type** attribute or an **element** attribute that specifies its data type. The data type attributes are listed in Table 3.1, "Part data type attributes".

Table 3.1. Part data type attributes

Attribute	Description
element="*elem_name*"	The data type of the part is defined by an element called *elem_name*.
type="*type_name*"	The data type of the part is defined by a type called *type_name*.

Messages are allowed to reuse part names. For instance, if a method has a parameter, ***foo***, that is passed by reference or is an in/out, it can be a part in both the request message and the response message, as shown in Example 3.1, “Reused part”.

Example 3.1. Reused part

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
<message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
<message>
```

EXAMPLE

For example, imagine you had a server that stored personal information and provided a method that returned an employee’s data based on the employee's ID number. The method signature for looking up the data is similar to Example 3.2, “personalInfo lookup method” .

Example 3.2. personalInfo lookup method

```
personalInfo lookup(long empId)
```

This method signature can be mapped to the RPC style WSDL fragment shown in Example 3.3, “RPC WSDL message definitions”.

Example 3.3. RPC WSDL message definitions

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int"/>
<message/>
<message name="personalLookupResponse>
  <part name="return" element="xsd1:personalInfo"/>
<message/>
```

It can also be mapped to the wrapped document style WSDL fragment shown in Example 3.4, “Wrapped document WSDL message definitions”.

Example 3.4. Wrapped document WSDL message definitions

```
<types>
  <schema ... >
  ...
  <element name="personalLookup">
    <complexType>
      <sequence>
        <element name="empID" type="xsd:int" />
      </sequence>
    </complexType>
```

```
    </element>
    <element name="personalLookupResponse">
      <complexType>
        <sequence>
          <element name="return" type="personalInfo" />
        </sequence>
      </complexType>
    </element>
    </schema>
  </types>
  <message name="personalLookupRequest">
    <part name="empId" element="xsd1:personalLookup"/>
  <message/>
  <message name="personalLookupResponse">
    <part name="return" element="xsd1:personalLookupResponse"/>
  <message/>
```

CHAPTER 4. DEFINING YOUR LOGICAL INTERFACES

Abstract

Logical service interfaces are defined using the **portType** element.

Logical service interfaces are defined using the WSDL **portType** element. The **portType** element is a collection of abstract operation definitions. Each operation is defined by the input, output, and fault messages used to complete the transaction the operation represents. When code is generated to implement the service interface defined by a **portType** element, each operation is converted into a method containing the parameters defined by the input, output, and fault messages specified in the contract.

PROCESS

To define a logical interface in a WSDL contract you must do the following:

1. Create a **portType** element to contain the interface definition and give it a unique name. See the section called “Port types”.
2. Create an **operation** element for each operation defined in the interface. See the section called “Operations”.
3. For each operation, specify the messages used to represent the operation’s parameter list, return type, and exceptions. See the section called “Operation messages”.

PORT TYPES

A WSDL **portType** element is the root element in a logical interface definition. While many Web service implementations map **portType** elements directly to generated implementation objects, a logical interface definition does not specify the exact functionality provided by the the implemented service. For example, a logical interface named **ticketSystem** can result in an implementation that either sells concert tickets or issues parking tickets.

The **portType** element is the unit of a WSDL document that is mapped into a binding to define the physical data used by an endpoint exposing the defined service.

Each **portType** element in a WSDL document must have a unique name, which is specified using the **name** attribute, and is made up of a collection of operations, which are described in **operation** elements. A WSDL document can describe any number of port types.

OPERATIONS

Logical operations, defined using WSDL **operation** elements, define the interaction between two endpoints. For example, a request for a checking account balance and an order for a gross of widgets can both be defined as operations.

Each operation defined within a **portType** element must have a unique name, specified using the **name** attribute. The **name** attribute is required to define an operation.

OPERATION MESSAGES

Logical operations are made up of a set of elements representing the logical messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in Table 4.1, "Operation message elements".

Table 4.1. Operation message elements

Element	Description
`input`	Specifies the message the client endpoint sends to the service provider when a request is made. The parts of this message correspond to the input parameters of the operation.
`output`	Specifies the message that the service provider sends to the client endpoint in response to a request. The parts of this message correspond to any operation parameters that can be changed by the service provider, such as values passed by reference. This includes the return value of the operation.
`fault`	Specifies a message used to communicate an error condition between the endpoints.

An operation is required to have at least one `input` or one `output` element. An operation can have both `input` and `output` elements, but it can only have one of each. Operations are not required to have any `fault` elements, but can, if required, have any number of `fault` elements.

The elements have the two attributes listed in Table 4.2, "Attributes of the input and output elements".

Table 4.2. Attributes of the input and output elements

Attribute	Description
`name`	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
`message`	Specifies the abstract message that describes the data being sent or received. The value of the `message` attribute must correspond to the `name` attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the `name` attribute for all `input` and `output` elements; WSDL provides a default naming scheme based on the enclosing operation's name. If only one element is used in the operation, the element name defaults to the name of the operation. If both an `input` and an `output` element are used, the element name defaults to the name of the operation with either `Request` or `Response` respectively appended to the name.

RETURN VALUES

Because the **operation** element is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value it will be mapped into the **output** element as the last part of that message.

EXAMPLE

For example, you might have an interface similar to the one shown in Example 4.1, "personalInfo lookup interface".

Example 4.1. personalInfo lookup interface

```
interface personalInfoLookup
{
  personalInfo lookup(in int empID)
  raises(idNotFound);
}
```

This interface can be mapped to the port type in Example 4.2, "personalInfo lookup port type".

Example 4.2. personalInfo lookup port type

```
<message name="personalLookupRequest">
  <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalLookupResponse"/>
<message/>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound"/>
<message/>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest"/>
    <output name="return" message="personalLookupResponse"/>
    <fault name="exception" message="idNotFoundException"/>
  </operation>
</portType>
```

PART II. WEB SERVICES BINDINGS

Abstract

This part describes how to add Apache CXF bindings to a WSDL document.

CHAPTER 5. UNDERSTANDING BINDINGS IN WSDL

Abstract

Bindings map the logical messages used to define a service into a concrete payload format that can be transmitted and received by an endpoint.

OVERVIEW

Bindings provide a bridge between the logical messages used by a service to a concrete data format that an endpoint uses in the physical world. They describe how the logical messages are mapped into a payload format that is used on the wire by an endpoint. It is within the bindings that details such as parameter order, concrete data types, and return values are specified. For example, the parts of a message can be reordered in a binding to reflect the order required by an RPC call. Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.

PORT TYPES AND BINDINGS

Port types and bindings are directly related. A port type is an abstract definition of a set of interactions between two logical services. A binding is a concrete definition of how the messages used to implement the logical services will be instantiated in the physical world. Each binding is then associated with a set of network details that finish the definition of one endpoint that exposes the logical service defined by the port type.

To ensure that an endpoint defines only a single service, WSDL requires that a binding can only represent a single port type. For example, if you had a contract with two port types, you could not write a single binding that mapped both of them into a concrete data format. You would need two bindings.

However, WSDL allows for a port type to be mapped to several bindings. For example, if your contract had a single port type, you could map it into two or more bindings. Each binding could alter how the parts of the message are mapped or they could specify entirely different payload formats for the message.

THE WSDL ELEMENTS

Bindings are defined in a contract using the WSDL **`binding`** element. The binding element has a single attribute, **`name`**, that specifies a unique name for the binding. The value of this attribute is used to associate the binding with an endpoint as discussed in Chapter 4, *Defining Your Logical Interfaces*.

The actual mappings are defined in the children of the **`binding`** element. These elements vary depending on the type of payload format you decide to use. The different payload formats and the elements used to specify their mappings are discussed in the following chapters.

ADDING TO A CONTRACT

Apache CXF provides command line tools that can generate bindings for predefined service interfaces.

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different types of bindings work.

You can also add a binding to a contract using any text editor. When hand editing a contract, you are responsible for ensuring that the contract is valid.

SUPPORTED BINDINGS

Apache CXF supports the following bindings:

- SOAP 1.1
- SOAP 1.2
- CORBA
- Pure XML

CHAPTER 6. USING SOAP 1.1 MESSAGES

Abstract

Apache CXF provides a tool to generate a SOAP 1.1 binding which does not use any SOAP headers. However, you can add SOAP headers to your binding using any text or XML editor.

6.1. ADDING A SOAP 1.1 BINDING

NOTE

To use **wsdl2soap** you will need to download the Apache CXF distribution.

Using wsdl2soap

To generate a SOAP 1.1 binding using **wsdl2soap** use the following command:

wsdl2soap { -i *port-type-name* } [-b *binding-name*] [-d *output-directory*] [-o *output-file*] [-n *soap-body-namespace*] [-style (document/rpc)] [-use (literal/encoded)] [-v] [[-verbose] | [-quiet]] *wsdlurl*

The command has the following options:

Option	Interpretation
-i *port-type-name*	Specifies the **portType** element for which a binding is generated.
wsdlurl	The path and name of the WSDL file containing the **portType** element definition.

The tool has the following optional arguments:

Option	Interpretation
-b *binding-name*	Specifies the name of the generated SOAP binding.
-d *output-directory*	Specifies the directory to place the generated WSDL file.
-o *output-file*	Specifies the name of the generated WSDL file.
-n *soap-body-namespace*	Specifies the SOAP body namespace when the style is RPC.
-style (**document/rpc**)	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is **document**.

Option	Interpretation
-use (**literal/encoded**)	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is **literal**.
-v	Displays the version number for the tool.
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.

The **-i** *port-type-name* and *wsdlurl* arguments are required. If the **-style rpc** argument is specified, the **-n** *soap-body-namspace* argument is also required. All other arguments are optional and may be listed in any order.

IMPORTANT

wsdl2soap does not support the generation of **document/encoded** SOAP bindings.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it is defined in a WSDL fragment similar to the one shown in Example 6.1, "Ordering System Interface".

Example 6.1. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
```

```
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

The SOAP binding generated for **orderWidgets** is shown in Example 6.2, "SOAP 1.1 Binding for **orderWidgets**".

Example 6.2. SOAP 1.1 Binding for orderWidgets

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the **document/literal** message style.

6.2. ADDING SOAP HEADERS TO A SOAP 1.1 BINDING

Overview

SOAP headers are defined by adding **soap:header** elements to your default SOAP 1.1 binding. The **soap:header** element is an optional child of the **input**, **output**, and **fault** elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many SOAP headers as needed.

Syntax

The syntax for defining a SOAP header is shown in Example 6.3, "SOAP Header Syntax". The **message** attribute of **soap:header** is the qualified name of the message from which the part being inserted into the header is taken. The **part** attribute is the name of the message part inserted into the SOAP header. Because SOAP headers are always document style, the WSDL message part inserted into the SOAP header must be defined using an element. Together the **message** and the **part** attributes fully describe the data to insert into the SOAP header.

Example 6.3. SOAP Header Syntax

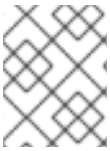
```
<binding name="headwig">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap:operation soapAction="" style="document"/>
      <input name="grain">
        <soap:body ... />
        <soap:header message="QName" part="partName"/>
      </input>
...
</binding>
```

As well as the mandatory **message** and **part** attributes, **soap:header** also supports the **namespace**, the **use**, and the **encodingStyle** attributes. These optional attributes function the same for **soap:header** as they do for **soap:body**.

Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would want to send information twice in the same message, the SOAP binding provides a means for specifying the message parts that are inserted into the SOAP body.

The **soap:body** element has an optional attribute, **parts**, that takes a space delimited list of part names. When **parts** is defined, only the message parts listed are inserted into the SOAP body. You can then insert the remaining parts into the SOAP header.



NOTE

When you define a SOAP header using parts of the parent message, Apache CXF automatically fills in the SOAP headers for you.

Example

Example 6.4, “SOAP 1.1 Binding with a SOAP Header” shows a modified version of the **orderWidgets** service shown in Example 6.1, “Ordering System Interface”. This version has been modified so that each order has an xsd:base64binary value placed in the SOAP header of the request and response. The SOAP header is defined as being the **keyVal** part from the **widgetKey** message. In this case you are responsible for adding the SOAP header to your application logic because it is not part of the input or output message.

Example 6.4. SOAP 1.1 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
```

```
<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

You can modify Example 6.4, “SOAP 1.1 Binding with a SOAP Header” so that the header value is a part of the input and output messages as shown in Example 6.5, “SOAP 1.1 Binding for orderWidgets with a SOAP Header”. In this case **keyVal** is a part of the input and output messages. In the **soap:body**

element's **parts** attribute specifies that **keyVal** cannot be inserted into the body. However, it is inserted into the SOAP header.

Example 6.5. SOAP 1.1 Binding for orderWidgets with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal" parts="numOrdered"/>
        <soap:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
        <soap:body use="literal" parts="bill"/>
        <soap:header message="tns:widgetOrderBill" part="keyVal"/>
      </output>
```

```
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

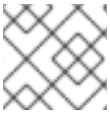
CHAPTER 7. USING SOAP 1.2 MESSAGES

Abstract

Apache CXF provides tools to generate a SOAP 1.2 binding which does not use any SOAP headers. You can add SOAP headers to your binding using any text or XML editor.

7.1. ADDING A SOAP 1.2 BINDING TO A WSDL DOCUMENT

Using wsdl2soap



NOTE

To use wsdl2soap you will need to download the Apache CXF distribution.

To generate a SOAP 1.2 binding using **wsdl2soap** use the following command:

wsdl2soap { -i *port-type-name* } [-b *binding-name*] { -soap12 } [-d *output-directory*] [-o *output-file*] [-n *soap-body-namespace*] [-style (document/rpc)] [-use (literal/encoded)] [-v] [[-verbose] | [-quiet]] *wsdlurl*

The tool has the following required arguments:

Option	Interpretation
-i *port-type-name*	Specifies the **portType** element for which a binding is generated.
-soap12	Specifies that the generated binding uses SOAP 1.2.
wsdlurl	The path and name of the WSDL file containing the **portType** element definition.

The tool has the following optional arguments:

Option	Interpretation
-b *binding-name*	Specifies the name of the generated SOAP binding.
-soap12	Specifies that the generated binding will use SOAP 1.2.
-d *output-directory*	Specifies the directory to place the generated WSDL file.
-o *output-file*	Specifies the name of the generated WSDL file.

Option	Interpretation
-n *soap-body-namespace*	Specifies the SOAP body namespace when the style is RPC.
-style (**document**/**rpc**)	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is **document**.
-use (**literal**/**encoded**)	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is **literal**.
-v	Displays the version number for the tool.
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.

The **-i** *port-type-name* and *wsdlurl* arguments are required. If the **-style rpc** argument is specified, the **-n** *soap-body-namspace* argument is also required. All other arguments are optional and can be listed in any order.

IMPORTANT

wsdl2soap does not support the generation of **document/encoded** SOAP 1.2 bindings.

Example

If your system has an interface that takes orders and offers a single operation to process the orders it is defined in a WSDL fragment similar to the one shown in Example 7.1, “Ordering System Interface”.

Example 7.1. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
```

```
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

The SOAP binding generated for **orderWidgets** is shown in Example 7.2, "SOAP 1.2 Binding for orderWidgets".

Example 7.2. SOAP 1.2 Binding for orderWidgets

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
      </input>
      <output name="bill">
        <wsoap12:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the **document/literal** message style.

7.2. ADDING HEADERS TO A SOAP 1.2 MESSAGE

Overview

SOAP message headers are defined by adding **soap12:header** elements to your SOAP 1.2 message. The **soap12:header** element is an optional child of the **input**, **output**, and **fault** elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many headers as needed.

Syntax

The syntax for defining a SOAP header is shown in Example 7.3, "SOAP Header Syntax".

Example 7.3. SOAP Header Syntax

```
<binding name="headwig">
  <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap12:operation soapAction="" style="documment"/>
      <input name="grain">
        <soap12:body ... />
        <soap12:header message="QName" part="partName"
                       use="literal|encoded"
                        encodingStyle="encodingURI"
                        namespace="namespaceURI" />
      </input>
...
</binding>
```

The **soap12:header** element's attributes are described in Table 7.1, "**soap12:header** Attributes".

Table 7.1. soap12:header Attributes

Attribute	Description
message	A required attribute specifying the qualified name of the message from which the part being inserted into the header is taken.
part	A required attribute specifying the name of the message part inserted into the SOAP header.
use	Specifies if the message parts are to be encoded using encoding rules. If set to **encoded** the message parts are encoded using the encoding rules specified by the value of the **encodingStyle** attribute. If set to **literal**, the message parts are defined by the schema types referenced.
encodingStyle	Specifies the encoding rules used to construct the message.
namespace	Defines the namespace to be assigned to the header element serialized with **use="encoded"**.

Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would send information twice in the same message, the SOAP 1.2 binding provides a means for specifying the message parts that are inserted into the SOAP body.

The **soap12:body** element has an optional attribute, **parts**, that takes a space delimited list of part names. When **parts** is defined, only the message parts listed are inserted into the body of the SOAP 1.2 message. You can then insert the remaining parts into the message's header.

NOTE

When you define a SOAP header using parts of the parent message, Apache CXF automatically fills in the SOAP headers for you.

Example

Example 7.4, "SOAP 1.2 Binding with a SOAP Header" shows a modified version of the **orderWidgets** service shown in Example 7.1, "Ordering System Interface". This version is modified so that each order has an xsd:base64binary value placed in the header of the request and the response. The header is defined as being the **keyVal** part from the **widgetKey** message. In this case you are responsible for adding the application logic to create the header because it is not part of the input or output message.

Example 7.4. SOAP 1.2 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
```

```
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

You can modify Example 7.4, “SOAP 1.2 Binding with a SOAP Header” so that the header value is a part of the input and output messages, as shown in Example 7.5, “SOAP 1.2 Binding for orderWidgets with a SOAP Header”. In this case **keyVal** is a part of the input and output messages. In the **soap12:body** elements the **parts** attribute specifies that **keyVal** should not be inserted into the body. However, it is inserted into the header.

Example 7.5. SOAP 1.2 Binding for orderWidgets with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
```

```
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal" parts="numOrdered"/>
        <soap12:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal" parts="bill"/>
        <soap12:header message="tns:widgetOrderBill" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

CHAPTER 8. SENDING BINARY DATA USING SOAP WITH ATTACHMENTS

Abstract

SOAP attachments provide a mechanism for sending binary data as part of a SOAP message. Using SOAP with attachments requires that you define your SOAP messages as MIME multipart messages.

OVERVIEW

SOAP messages generally do not carry binary data. However, the W3C SOAP 1.1 specification allows for using MIME multipart/related messages to send binary data in SOAP messages. This technique is called using SOAP with attachments. SOAP attachments are defined in the W3C's *SOAP Messages with Attachments Note*.

NAMESPACE

The WSDL extensions used to define the MIME multipart/related messages are defined in the namespace http://schemas.xmlsoap.org/wsdl/mime/.

In the discussion that follows, it is assumed that this namespace is prefixed with **mime**. The entry in the WSDL **definitions** element to set this up is shown in Example 8.1, "MIME Namespace Specification in a Contract".

Example 8.1. MIME Namespace Specification in a Contract

```
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

CHANGING THE MESSAGE BINDING

In a default SOAP binding, the first child element of the **input**, **output**, and **fault** elements is a **soap:body** element describing the body of the SOAP message representing the data. When using SOAP with attachments, the **soap:body** element is replaced with a **mime:multipartRelated** element.

NOTE

WSDL does not support using **mime:multipartRelated** for **fault** messages.

The **mime:multipartRelated** element tells Apache CXF that the message body is a multipart message that potentially contains binary data. The contents of the element define the parts of the message and their contents. **mime:multipartRelated** elements contain one or more **mime:part** elements that describe the individual parts of the message.

The first **mime:part** element must contain the **soap:body** element that would normally appear in a default SOAP binding. The remaining **mime:part** elements define the attachments that are being sent in the message.

DESCRIBING A MIME MULTIPART MESSAGE

MIME multipart messages are described using a **mime:multipartRelated** element that contains a number of **mime:part** elements. To fully describe a MIME multipart message you must do the following:

1. Inside the **input** or **output** message you are sending as a MIME multipart message, add a **mime:mulipartRelated** element as the first child element of the enclosing message.
2. Add a **mime:part** child element to the **mime:multipartRelated** element and set its **name** attribute to a unique string.
3. Add a **soap:body** element as the child of the **mime:part** element and set its attributes appropriately.

   TIP

   If the contract had a default SOAP binding, you can copy the **soap:body** element from the corresponding message from the default binding into the MIME multipart message.

4. Add another **mime:part** child element to the **mime:multipartReleated** element and set its **name** attribute to a unique string.
5. Add a **mime:content** child element to the **mime:part** element to describe the contents of this part of the message.

   To fully describe the contents of a MIME message part the **mime:content** element has the following attributes:

   Table 8.1. mime:content Attributes

Attribute	Description
part	Specifies the name of the WSDL message **part**, from the parent message definition, that is used as the content of this part of the MIME multipart message being placed on the wire.
type	The MIME type of the data in this message part. MIME types are defined as a type and a subtype using the syntax *type/ subtype*. There are a number of predefined MIME types such as **image/jpeg** and **text/plain**. The MIME types are maintained by the Internet Assigned Numbers Authority (IANA) and described in detail in *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*.

6. For each additional MIME part, repeat steps Step 4 and Step 5.

EXAMPLE

Example 8.2, "Contract using SOAP with Attachments" shows a WSDL fragment defining a service that stores X-rays in JPEG format. The image data, **xRay**, is stored as an xsd:base64binary and is packed into the MIME multipart message's second part, **imageData**. The remaining two parts of the input message, **patientName** and **patientNumber**, are sent in the first part of the MIME multipart image as part of the SOAP body.

Example 8.2. Contract using SOAP with Attachments

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="storRequest">
    <part name="patientName" type="xsd:string"/>
    <part name="patientNumber" type="xsd:int"/>
    <part name="xRay" type="xsd:base64Binary"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>

  <binding name="xRayStorageBinding" type="tns:xRayStorage">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="store">
      <soap:operation soapAction="" style="document"/>
      <input name="storRequest">
        <mime:multipartRelated>
          <mime:part name="bodyPart">
            <soap:body use="literal"/>
          </mime:part>
          <mime:part name="imageData">
            <mime:content part="xRay" type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </input>
      <output name="storResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="xRayStorageService">
    <port binding="tns:xRayStorageBinding" name="xRayStoragePort">
```

```
      <soap:address location="http://localhost:9000"/>
    </port>
  </service>
</definitions>
```

CHAPTER 9. SENDING BINARY DATA WITH SOAP MTOM

Abstract

SOAP Message Transmission Optimization Mechanism (MTOM) replaces SOAP with attachments as a mechanism for sending binary data as part of an XML message. Using MTOM with Apache CXF requires adding the correct schema types to a service's contract and enabling the MTOM optimizations.

SOAP Message Transmission Optimization Mechanism (MTOM) specifies an optimized method for sending binary data as part of a SOAP message. Unlike SOAP with Attachments, MTOM requires the use of XML-binary Optimized Packaging (XOP) packages for transmitting binary data. Using MTOM to send binary data does not require you to fully define the MIME Multipart/Related message as part of the SOAP binding. It does, however, require that you do the following:

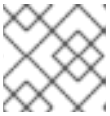
1. Annotate the data that you are going to send as an attachment.

   You can annotate either your WSDL or the Java class that implements your data.

2. Enable the runtime's MTOM support.

   This can be done either programmatically or through configuration.

3. Develop a **`DataHandler`** for the data being passed as an attachment.



NOTE

Developing **`DataHandlers`** is beyond the scope of this book.

9.1. ANNOTATING DATA TYPES TO USE MTOM

Overview

In WSDL, when defining a data type for passing along a block of binary data, such as an image file or a sound file, you define the element for the data to be of type xsd:base64Binary. By default, any element of type xsd:base64Binary results in the generation of a byte[] which can be serialized using MTOM. However, the default behavior of the code generators does not take full advantage of the serialization.

In order to fully take advantage of MTOM you must add annotations to either your service's WSDL document or the JAXB class that implements the binary data structure. Adding the annotations to the WSDL document forces the code generators to generate streaming data handlers for the binary data. Annotating the JAXB class involves specifying the proper content types and might also involve changing the type specification of the field containing the binary data.

WSDL first

Example 9.1, “Message for MTOM” shows a WSDL document for a Web service that uses a message which contains one string field, one integer field, and a binary field. The binary field is intended to carry a large image file, so it is not appropriate to send it as part of a normal SOAP message.

Example 9.1. Message for MTOM

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
```

```
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:xsd1="http://mediStor.org/types/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>
    <schema targetNamespace="http://mediStor.org/types/"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary" />
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
    </schema>
  </types>

  <message name="storRequest">
    <part name="record" element="xsd1:xRay"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>

  <binding name="xRayStorageSOAPBinding" type="tns:xRayStorage">
    <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="store">
      <soap12:operation soapAction="" style="document"/>
      <input name="storRequest">
        <soap12:body use="literal"/>
      </input>
      <output name="storResponse">
        <soap12:body use="literal"/>
      </output>
    </operation>
  </binding>
  ...
</definitions>
```

If you want to use MTOM to send the binary part of the message as an optimized attachment you must add the **xmime:expectedContentTypes** attribute to the element containing the binary data. This attribute is defined in the http://www.w3.org/2005/05/xmlmime namespace and specifies the MIME types that the element is expected to contain. You can specify a comma separated list of MIME types.

The setting of this attribute changes how the code generators create the JAXB class for the data. For most MIME types, the code generator creates a **`DataHandler`**. Some MIME types, such as those for images, have defined mappings.

NOTE

The MIME types are maintained by the Internet Assigned Numbers Authority(IANA) and are described in detail in *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*.

TIP

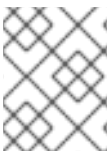
For most uses you specify **`application/octet-stream`**.

Example 9.2, "Binary Data for MTOM" shows how you can modify xRayType from Example 9.1, "Message for MTOM" for using MTOM.

Example 9.2. Binary Data for MTOM

```
...
  <types>
    <schema targetNamespace="http://mediStor.org/types/"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary"
                   xmime:expectedContentTypes="application/octet-
stream"/>
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
    </schema>
  </types>
...
```

The generated JAXB class generated for xRayType no longer contains a byte[]. Instead the code generator sees the **`xmime:expectedContentTypes`** attribute and generates a **`DataHandler`** for the imageData field.



NOTE

You do not need to change the **`binding`** element to use MTOM. The runtime makes the appropriate changes when the data is sent.

Java first

If you are doing Java first development you can make your JAXB class MTOM ready by doing the following:

1. Make sure the field holding the binary data is a **`DataHandler`**.
2. Add the **`@XmlMimeType()`** annotation to the field containing the data you want to stream as an MTOM attachment.

Example 9.3, "JAXB Class for MTOM" shows a JAXB class annotated for using MTOM.

Example 9.3. JAXB Class for MTOM

```
@XmlType
public class XRayType {
    protected String patientName;
    protected int patientNumber;
    @XmlMimeType("application/octet-stream")
    protected DataHandler imageData;
  ...
}
```

9.2. ENABLING MTOM

By default the Apache CXF runtime does not enable MTOM support. It sends all binary data as either part of the normal SOAP message or as an unoptimized attachment. You can activate MTOM support either programmatically or through the use of configuration.

9.2.1. Using JAX-WS APIs

Overview

Both service providers and consumers must have the MTOM optimizations enabled. The JAX-WS APIs offer different mechanisms for each type of endpoint.

Service provider

If you published your service provider using the JAX-WS APIs you enable the runtime's MTOM support as follows:

1. Access the **`Endpoint`** object for your published service.

   The easiest way to access the **`Endpoint`** object is when you publish the endpoint. For more information see Chapter 30, *Publishing a Service*.

2. Get the SOAP binding from the **`Endpoint`** using its **`getBinding()`** method, as shown in Example 9.4, "Getting the SOAP Binding from an Endpoint" .

   Example 9.4. Getting the SOAP Binding from an Endpoint

   ```
   // Endpoint ep is declared previously
   SOAPBinding binding = (SOAPBinding)ep.getBinding();
   ```

You must cast the returned binding object to a **SOAPBinding** object to access the MTOM property.

3. Set the binding's MTOM enabled property to **true** using the binding's **setMTOMEnabled()** method, as shown in Example 9.5, "Setting a Service Provider's MTOM Enabled Property" .

Example 9.5. Setting a Service Provider's MTOM Enabled Property

```
binding.setMTOMEnabled(true);
```

Consumer

To MTOM enable a JAX-WS consumer you must do the following:

1. Cast the consumer's proxy to a **BindingProvider** object.

TIP

For information on getting a consumer proxy see Chapter 24, *Developing a Consumer Without a WSDL Contract* or Chapter 27, *Developing a Consumer From a WSDL Contract*.

2. Get the SOAP binding from the **BindingProvider** using its **getBinding()** method, as shown in Example 9.6, "Getting a SOAP Binding from a **BindingProvider**".

Example 9.6. Getting a SOAP Binding from a BindingProvider

```
// BindingProvider bp declared previously
SOAPBinding binding = (SOAPBinding)bp.getBinding();
```

3. Set the bindings MTOM enabled property to **true** using the binding's **setMTOMEnabled()** method, as shown in Example 9.7, "Setting a Consumer's MTOM Enabled Property" .

Example 9.7. Setting a Consumer's MTOM Enabled Property

```
binding.setMTOMEnabled(true);
```

9.2.2. Using configuration

Overview

If you publish your service using XML, such as when deploying to a container, you can enable your endpoint's MTOM support in the endpoint's configuration file. For more information on configuring endpoint's see Part IV, "Configuring Web Service Endpoints" .

Procedure

The MTOM property is set inside the **jaxws:endpoint** element for your endpoint. To enable MTOM do the following:

1. Add a **jaxws:property** child element to the endpoint's **jaxws:endpoint** element.
2. Add a **entry** child element to the **jaxws:property** element.
3. Set the **entry** element's **key** attribute to **mtom-enabled**.
4. Set the **entry** element's **value** attribute to **true**.

Example

Example 9.8, "Configuration for Enabling MTOM" shows an endpoint that is MTOM enabled.

Example 9.8. Configuration for Enabling MTOM

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://cxf.apache.org/jaxws
http://cxf.apache.org/schema/jaxws.xsd">

  <jaxws:endpoint id="xRayStorage"
                  implementor="demo.spring.xRayStorImpl"
                  address="http://localhost/xRayStorage">
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:endpoint>
</beans>
```

CHAPTER 10. USING XML DOCUMENTS

Abstract

The pure XML payload format provides an alternative to the SOAP binding by allowing services to exchange data using straight XML documents without the overhead of a SOAP envelope.

XML BINDING NAMESPACE

The extensions used to describe XML format bindings are defined in the namespace http://cxf.apache.org/bindings/xformat. Apache CXF tools use the prefix **xformat** to represent the XML binding extensions. Add the following line to your contracts:

```
xmlns:xformat="http://cxf.apache.org/bindings/xformat"
```

HAND EDITING

To map an interface to a pure XML payload format do the following:

1. Add the namespace declaration to include the extensions defining the XML binding. See the section called “XML binding namespace”.
2. Add a standard WSDL **binding** element to your contract to hold the XML binding, give the binding a unique **name**, and specify the name of the WSDL **portType** element that represents the interface being bound.
3. Add an **xformat:binding** child element to the **binding** element to identify that the messages are being handled as pure XML documents without SOAP envelopes.
4. Optionally, set the **xformat:binding** element's **rootNode** attribute to a valid QName. For more information on the effect of the **rootNode** attribute see the section called “XML messages on the wire”.
5. For each operation defined in the bound interface, add a standard WSDL **operation** element to hold the binding information for the operation's messages.
6. For each operation added to the binding, add the **input**, **output**, and **fault** children elements to represent the messages used by the operation.

   These elements correspond to the messages defined in the interface definition of the logical operation.
7. Optionally add an **xformat:body** element with a valid **rootNode** attribute to the added **input**, **output**, and **fault** elements to override the value of **rootNode** set at the binding level.

NOTE

If any of your messages have no parts, for example the output message for an operation that returns void, you must set the **rootNode** attribute for the message to ensure that the message written on the wire is a valid, but empty, XML document.

XML MESSAGES ON THE WIRE

When you specify that an interface's messages are to be passed as XML documents, without a SOAP envelope, you must take care to ensure that your messages form valid XML documents when they are written on the wire. You also need to ensure that non-Apache CXF participants that receive the XML documents understand the messages generated by Apache CXF.

A simple way to solve both problems is to use the optional **rootNode** attribute on either the global **xformat:binding** element or on the individual message's **xformat:body** elements. The **rootNode** attribute specifies the QName for the element that serves as the root node for the XML document generated by Apache CXF. When the **rootNode** attribute is not set, Apache CXF uses the root element of the message part as the root element when using doc style messages, or an element using the message part name as the root element when using rpc style messages.

For example, if the **rootNode** attribute is not set the message defined in Example 10.1, "Valid XML Binding Message" would generate an XML document with the root element **lineNumber**.

Example 10.1. Valid XML Binding Message

```
<type ... >
  ...
  <element name="operatorID" type="xsd:int"/>
  ...
</types>
<message name="operator">
  <part name="lineNumber" element="ns1:operatorID"/>
</message>
```

For messages with one part, Apache CXF will always generate a valid XML document even if the **rootNode** attribute is not set. However, the message in Example 10.2, "Invalid XML Binding Message" would generate an invalid XML document.

Example 10.2. Invalid XML Binding Message

```
<types>
  ...
  <element name="pairName" type="xsd:string"/>
  <element name="entryNum" type="xsd:int"/>
  ...
</types>

<message name="matildas">
  <part name="dancing" element="ns1:pairName"/>
  <part name="number" element="ns1:entryNum"/>
</message>
```

Without the **rootNode** attribute specified in the XML binding, Apache CXF will generate an XML document similar to Example 10.3, "Invalid XML Document" for the message defined in Example 10.2, "Invalid XML Binding Message". The generated XML document is invalid because it has two root elements: **pairName** and **entryNum**.

Example 10.3. Invalid XML Document

```
<pairName>
  Fred&Linda
</pairName>
<entryNum>
  123
</entryNum>
```

If you set the **rootNode** attribute, as shown in Example 10.4, "XML Binding with rootNode set" Apache CXF will wrap the elements in the specified root element. In this example, the **rootNode** attribute is defined for the entire binding and specifies that the root element will be named entrants.

Example 10.4. XML Binding with rootNode set

```
<portType name="danceParty">
  <operation name="register">
    <input message="tns:matildas" name="contestant"/>
  </operation>
</portType>

<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered"/>
</binding>
```

An XML document generated from the input message would be similar to Example 10.5, "XML Document generated using the rootNode attribute". Notice that the XML document now only has one root element.

Example 10.5. XML Document generated using the rootNode attribute

```
<entrants>
  <pairName>
    Fred&Linda
  <entryNum>
    123
  </entryNum>
</entrants>
```

OVERRIDING THE BINDING'S ROOTNODE ATTRIBUTE SETTING

You can also set the **rootNode** attribute for each individual message, or override the global setting for a particular message, by using the **xformat:body** element inside of the message binding. For example, if you wanted the output message defined in Example 10.4, "XML Binding with rootNode set" to have a different root element from the input message, you could override the binding's root element as shown in Example 10.6, "Using **xformat:body**".

Example 10.6. Using `xformat:body`

```
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered">
      <xformat:body rootNode="entryStatus" />
    </output>
  </operation>
</binding>
```

PART III. WEB SERVICES TRANSPORTS

Abstract

This part describes how to add Apache CXF transports to a WSDL document.

CHAPTER 11. UNDERSTANDING HOW ENDPOINTS ARE DEFINED IN WSDL

Abstract

Endpoints represent an instantiated service. They are defined by combining a binding and the networking details used to expose the endpoint.

OVERVIEW

An endpoint can be thought of as a physical manifestation of a service. It combines a binding, which specifies the physical representation of the logical data used by a service, and a set of networking details that define the physical connection details used to make the service contactable by other endpoints.

ENDPOINTS AND SERVICES

In the same way a binding can only map a single interface, an endpoint can only map to a single service. However, a service can be manifested by any number of endpoints. For example, you could define a ticket selling service that was manifested by four different endpoints. However, you could not have a single endpoint that manifested both a ticket selling service and a widget selling service.

THE WSDL ELEMENTS

Endpoints are defined in a contract using a combination of the WSDL **`service`** element and the WSDL **`port`** element. The **`service`** element is a collection of related **`port`** elements. The **`port`** elements define the actual endpoints.

The WSDL **`service`** element has a single attribute, **`name`**, that specifies a unique name. The **`service`** element is used as the parent element of a collection of related **`port`** elements. WSDL makes no specification about how the **`port`** elements are related. You can associate the **`port`** elements in any manner you see fit.

The WSDL **`port`** element has a single attribute, **`binding`**, that specifies the binding used by the endpoint. The **`port`** element is the parent element of the elements that specify the actual transport details used by the endpoint. The elements used to specify the transport details are discussed in the following sections.

ADDING ENDPOINTS TO A CONTRACT

Apache CXF provides command line tools that can generated endpoints for predefined service interface and binding combinations.

The tools will add the proper elements to your contract for you. However, it is recommended that you have some knowledge of how the different transports used in defining an endpoint work.

You can also add an endpoint to a contract using any text editor. When you hand edit a contract, you are responsible for ensuring that the contract is valid.

SUPPORTED TRANSPORTS

Endpoint definitions are built using extensions defined for each of the transports Apache CXF supports. This includes the following transports:

- HTTP
- CORBA
- Java Messaging Service

CHAPTER 12. USING HTTP

Abstract

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Because of these factors it is the assumed transport for most WS-* specifications and is integral to RESTful architectures.

12.1. ADDING A BASIC HTTP ENDPOINT

Alternative HTTP runtimes

Apache CXF supports the following alternative HTTP runtime implementations:

- Jetty, which is described in detail in Section 12.4, "Configuring the Jetty Runtime".
- Netty, which is described in detail in Section 12.5, "Configuring the Netty Runtime".

Netty HTTP URL

Normally, a HTTP endpoint uses whichever HTTP runtime is included on the classpath (either Jetty or Netty). If both the Jetty runtime and Netty runtime are included on the classpath, however, you need to specify explicitly when you want to use the Netty runtime, because the Jetty runtime will be used by default.

In the case where more than one HTTP runtime is available on the classpath, you can select the Netty runtime by specifying the endpoint URL to have the following format:

```
netty://http://RestOfURL
```

Payload types

There are three ways of specifying an HTTP endpoint's address depending on the payload format you are using.

- SOAP 1.1 uses the standardized **`soap:address`** element.
- SOAP 1.2 uses the **`soap12:address`** element.
- All other payload formats use the **`http:address element`**.

SOAP 1.1

When you are sending SOAP 1.1 messages over HTTP you must use the SOAP 1.1 **`address`** element to specify the endpoint's address. It has one attribute, **`location`**, that specifies the endpoint's address as a URL. The SOAP 1.1 **`address`** element is defined in the namespace http://schemas.xmlsoap.org/wsdl/soap/.

Example 12.1, "SOAP 1.1 Port Element" shows a **`port`** element used to send SOAP 1.1 messages over HTTP.

Example 12.1. SOAP 1.1 Port Element

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
  ...
  <service name="SOAP11Service">
    <port binding="SOAP11Binding" name="SOAP11Port">
      <soap:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

SOAP 1.2

When you are sending SOAP 1.2 messages over HTTP you must use the SOAP 1.2 **address** element to specify the endpoint's address. It has one attribute, **location**, that specifies the endpoint's address as a URL. The SOAP 1.2 **address** element is defined in the namespace http://schemas.xmlsoap.org/wsdl/soap12/.

Example 12.2, "SOAP 1.2 Port Element" shows a **port** element used to send SOAP 1.2 messages over HTTP.

Example 12.2. SOAP 1.2 Port Element

```
<definitions ...
             xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" ...
>
  <service name="SOAP12Service">
    <port binding="SOAP12Binding" name="SOAP12Port">
      <soap12:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

Other messages types

When your messages are mapped to any payload format other than SOAP you must use the HTTP **address** element to specify the endpoint's address. It has one attribute, **location**, that specifies the endpoint's address as a URL. The HTTP **address** element is defined in the namespace http://schemas.xmlsoap.org/wsdl/http/.

Example 12.3, "HTTP Port Element" shows a **port** element used to send an XML message.

Example 12.3. HTTP Port Element

```
<definitions ...
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" ... >
  <service name="HTTPService">
    <port binding="HTTPBinding" name="HTTPPort">
```

```
      <http:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

12.2. CONFIGURING A CONSUMER

HTTP consumer endpoints can specify a number of HTTP connection attributes including whether the endpoint automatically accepts redirect responses, whether the endpoint can use chunking, whether the endpoint will request a keep-alive, and how the endpoint interacts with proxies. In addition to the HTTP connection properties, an HTTP consumer endpoint can specify how it is secured.

A consumer endpoint can be configured using two mechanisms:

- Configuration
- WSDL

12.2.1. Using Configuration

Namespace

The elements used to configure an HTTP consumer endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration. It is commonly referred to using the prefix **`http-conf`**. In order to use the HTTP configuration elements you must add the lines shown in Example 12.4, "HTTP Consumer Configuration Namespace" to the **beans** element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the **`xsi:schemaLocation`** attribute.

Example 12.4. HTTP Consumer Configuration Namespace

```
<beans ...
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"
       ...
       xsi:schemaLocation="...

http://cxf.apache.org/transports/http/configuration

http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           ...">
```

Jetty runtime or Netty runtime

You can use the elements from the **`http-conf`** namespace to configure either the Jetty runtime or the Netty runtime.

The `conduit` element

You configure an HTTP consumer endpoint using the **http-conf:conduit** element and its children. The **http-conf:conduit** element takes a single attribute, **name**, that specifies the WSDL **port** element corresponding to the endpoint. The value for the **name** attribute takes the form *portQName*.**http-conduit**. Example 12.5, "**http-conf:conduit** Element" shows the **http-conf:conduit** element that would be used to add configuration for an endpoint that is specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> when the endpoint's target namespace is http://widgets.widgetvendor.net.

Example 12.5. http-conf:conduit Element

```
...
  <http-conf:conduit name="
{http://widgets/widgetvendor.net}widgetSOAPPort.http-conduit">
    ...
  </http-conf:conduit>
...
```

The **http-conf:conduit** element has child elements that specify configuration information. They are described in Table 12.1, "Elements Used to Configure an HTTP Consumer Endpoint" .

Table 12.1. Elements Used to Configure an HTTP Consumer Endpoint

Element	Description
http-conf:client	Specifies the HTTP connection properties such as timeouts, keep-alive requests, content types, etc. See the section called "The **client** element".
http-conf:authorization	Specifies the parameters for configuring the basic authentication method that the endpoint uses preemptively. The preferred approach is to supply a Basic Authentication Supplier object.
http-conf:proxyAuthorization	Specifies the parameters for configuring basic authentication against outgoing HTTP proxy servers.
http-conf:tlsClientParameters	Specifies the parameters used to configure SSL/TLS.
http-conf:basicAuthSupplier	Specifies the bean reference or class name of the object that supplies the basic authentication information used by the endpoint, either preemptively or in response to a **401** HTTP challenge.
http-conf:trustDecider	Specifies the bean reference or class name of the object that checks the HTTP(S) **URLConnection** object to establish trust for a connection with an HTTPS service provider before any information is transmitted.

The `client` element

The `http-conf:client` element is used to configure the non-security properties of a consumer endpoint's HTTP connection. Its attributes, described in Table 12.2, "HTTP Consumer Configuration Attributes", specify the connection's properties.

Table 12.2. HTTP Consumer Configuration Attributes

Attribute	Description
`ConnectionTimeout`	Specifies the amount of time, in milliseconds, that the consumer attempts to establish a connection before it times out. The default is `30000`. `0` specifies that the consumer will continue to send the request indefinitely.
`ReceiveTimeout`	Specifies the amount of time, in milliseconds, that the consumer will wait for a response before it times out. The default is `30000`. `0` specifies that the consumer will wait indefinitely.
`AutoRedirect`	Specifies if the consumer will automatically follow a server issued redirection. The default is `false`.
`MaxRetransmits`	Specifies the maximum number of times a consumer will retransmit a request to satisfy a redirect. The default is `-1` which specifies that unlimited retransmissions are allowed.
`AllowChunking`	Specifies whether the consumer will send requests using chunking. The default is `true` which specifies that the consumer will use chunking when sending requests. Chunking cannot be used if either of the following are true: • `http-conf:basicAuthSupplier` is configured to provide credentials preemptively. • `AutoRedirect` is set to `true`. In both cases the value of `AllowChunking` is ignored and chunking is disallowed.
`Accept`	Specifies what media types the consumer is prepared to handle. The value is used as the value of the HTTP Accept property. The value of the attribute is specified using multipurpose internet mail extensions (MIME) types.

Attribute	Description
`AcceptLanguage`	Specifies what language (for example, American English) the consumer prefers for the purpose of receiving a response. The value is used as the value of the HTTP AcceptLanguage property. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, en-US represents American English.
`AcceptEncoding`	Specifies what content encodings the consumer is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP AcceptEncoding property.
`ContentType`	Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType property. The default is **`text/xml`**. For web services, this should be set to **`text/xml`**. If the client is sending HTML form data to a CGI script, this should be set to **`application/x-www-form-urlencoded`**. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to **`application/octet-stream`**.
`Host`	Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP Host property. This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).

Attribute	Description
`Connection`	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • **`Keep-Alive`** – Specifies that the consumer wants the connection kept open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it. • **`close`**(default) – Specifies that the connection to the server is closed after each request/response sequence.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a consumer to a service provider. See Section 12.2.3, "Consumer Cache Control Directives".
`Cookie`	Specifies a static cookie to be sent with all requests.
`BrowserType`	Specifies information about the browser from which the request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent*. Some servers optimize based on the client that is sending the request.
`Referer`	Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP Referer property. This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. If the **`AutoRedirect`** attribute is set to **`true`** and the request is redirected, any value specified in the **`Referer`** attribute is overridden. The value of the HTTP Referer property is set to the URL of the service that redirected the consumer's original request.

Attribute	Description
DecoupledEndpoint	Specifies the URL of a decoupled endpoint for the receipt of responses over a separate provider->consumer connection. For more information on using decoupled endpoints see, Section 12.6, "Using the HTTP Transport in Decoupled Mode". You must configure both the consumer endpoint and the service provider endpoint to use WS-Addressing for the decoupled endpoint to work.
ProxyServer	Specifies the URL of the proxy server through which requests are routed.
ProxyServerPort	Specifies the port number of the proxy server through which requests are routed.
ProxyServerType	Specifies the type of proxy server used to route requests. Valid values are: • **HTTP**(default) • **SOCKS**

Example

Example 12.6, "HTTP Consumer Endpoint Configuration" shows the configuration of an HTTP consumer endpoint that wants to keep its connection to the provider open between requests, that will only retransmit requests once per invocation, and that cannot use chunking streams.

Example 12.6. HTTP Consumer Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"

xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration

http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:conduit name="
{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
    <http-conf:client Connection="Keep-Alive"
                      MaxRetransmits="1"
                      AllowChunking="false" />
  </http-conf:conduit>
</beans>
```

More information

For more information on HTTP conduits see Appendix B, *Conduits*.

12.2.2. Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP consumer endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration. It is commonly referred to using the prefix **`http-conf`**. In order to use the HTTP configuration elements you must add the line shown in Example 12.7, "HTTP Consumer WSDL Element's Namespace" to the **`definitions`** element of your endpoint's WSDL document.

Example 12.7. HTTP Consumer WSDL Element's Namespace

```
<definitions ...
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"
```

Jetty runtime or Netty runtime

You can use the elements from the **`http-conf`** namespace to configure either the Jetty runtime or the Netty runtime.

The `client` element

The **`http-conf:client`** element is used to specify the connection properties of an HTTP consumer in a WSDL document. The **`http-conf:client`** element is a child of the WSDL **`port`** element. It has the same attributes as the **`client`** element used in the configuration file. The attributes are described in Table 12.2, "HTTP Consumer Configuration Attributes".

Example

Example 12.8, "WSDL to Configure an HTTP Consumer Endpoint" shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it does not interact with caches.

Example 12.8. WSDL to Configure an HTTP Consumer Endpoint

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:client CacheControl="no-cache" />
  </port>
</service>
```

12.2.3. Consumer Cache Control Directives

Table 12.3, "**`http-conf:client`** Cache Control Directives" lists the cache control directives supported by an HTTP consumer.

Table 12.3. `http-conf:client` Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store either any part of a response or any part of the request that invoked it.
max-age	The consumer can accept a response whose age is no greater than the specified time in seconds.
max-stale	The consumer can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the consumer can still accept that response. If no value is assigned, the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that is still fresh for at least the specified number of seconds indicated.
no-transform	Caches must not modify media type or location of the content in a response between a provider and a consumer.
only-if-cached	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

12.3. CONFIGURING A SERVICE PROVIDER

HTTP service provider endpoints can specify a number of HTTP connection attributes including if it will honor keep alive requests, how it interacts with caches, and how tolerant it is of errors in communicating with a consumer.

A service provider endpoint can be configured using two mechanisms:

- Configuration
- WSDL

12.3.1. Using Configuration

Namespace

The elements used to configure an HTTP provider endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration. It is commonly referred to using the prefix **http-conf**. In order to use the HTTP configuration elements you must add the lines shown in Example 12.9, "HTTP Provider Configuration Namespace" to the **beans** element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the **xsi:schemaLocation** attribute.

Example 12.9. HTTP Provider Configuration Namespace

```
<beans ...
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"
       ...
       xsi:schemaLocation="...

http://cxf.apache.org/transports/http/configuration

http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           ...">
```

Jetty runtime or Netty runtime

You can use the elements from the **http-conf** namespace to configure either the Jetty runtime or the Netty runtime.

The destination element

You configure an HTTP service provider endpoint using the **http-conf:destination** element and its children. The **http-conf:destination** element takes a single attribute, **name**, that specifies the WSDL **port** element that corresponds to the endpoint. The value for the **name** attribute takes the form *portQName*.**http-destination**. Example 12.10, "**http-conf:destination** Element" shows the **http-conf:destination** element that is used to add configuration for an endpoint that is specified by the WSDL fragment <port binding="widgetSOAPBinding" name="widgetSOAPPort> when the endpoint's target namespace is http://widgets.widgetvendor.net.

Example 12.10. http-conf:destination Element

```
...
  <http-conf:destination name="
{http://widgets/widgetvendor.net}widgetSOAPPort.http-destination">
    ...
  </http-conf:destination>
...
```

The **http-conf:destination** element has a number of child elements that specify configuration information. They are described in Table 12.4, "Elements Used to Configure an HTTP Service Provider Endpoint".

Table 12.4. Elements Used to Configure an HTTP Service Provider Endpoint

Element	Description
http-conf:server	Specifies the HTTP connection properties. See the section called "The **server** element".
http-conf:contextMatchStrategy	Specifies the parameters that configure the context match strategy for processing HTTP requests.
http-conf:fixedParameterOrder	Specifies whether the parameter order of an HTTP request handled by this destination is fixed.

The server element

The **http-conf:server** element is used to configure the properties of a service provider endpoint's HTTP connection. Its attributes, described in Table 12.5, "HTTP Service Provider Configuration Attributes", specify the connection's properties.

Table 12.5. HTTP Service Provider Configuration Attributes

Attribute	Description
ReceiveTimeout	Sets the length of time, in milliseconds, the service provider attempts to receive a request before the connection times out. The default is **30000**. **0** specifies that the provider will not timeout.
SuppressClientSendErrors	Specifies whether exceptions are to be thrown when an error is encountered on receiving a request. The default is **false**; exceptions are thrown on encountering errors.
SuppressClientReceiveErrors	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a consumer. The default is **false**; exceptions are thrown on encountering errors.
HonorKeepAlive	Specifies whether the service provider honors requests for a connection to remain open after a response has been sent. The default is **false**; keep-alive requests are ignored.

Attribute	Description
RedirectURL	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is set to **302** and the status description is set to **Object Moved**. The value is used as the value of the HTTP RedirectURL property.
CacheControl	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a service provider to a consumer. See Section 12.3.3, "Service Provider Cache Control Directives".
ContentLocation	Sets the URL where the resource being sent in a response is located.
ContentType	Specifies the media type of the information being sent in a response. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType location.
ContentEncoding	Specifies any additional content encodings that have been applied to the information being sent by the service provider. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include **zip**, **gzip**, **compress**, **deflate**, and **identity**. This value is used as the value of the HTTP ContentEncoding property. The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Apache CXF performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level.
ServerType	Specifies what type of server is sending the response. Values take the form ***program-name/version***; for example, **Apache/1.2.5**.

Example

Example 12.11, "HTTP Service Provider Endpoint Configuration" shows the configuration for an HTTP service provider endpoint that honors keep-alive requests and suppresses all communication errors.

Example 12.11. HTTP Service Provider Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"

xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration

http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:destination name="
{http://apache.org/hello_world_soap_http}SoapPort.http-destination">
    <http-conf:server SuppressClientSendErrors="true"
                      SuppressClientReceiveErrors="true"
                      HonorKeepAlive="true" />
  </http-conf:destination>
</beans>
```

12.3.2. Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP provider endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration. It is commonly referred to using the prefix **`http-conf`**. To use the HTTP configuration elements you must add the line shown in Example 12.12, "HTTP Provider WSDL Element's Namespace" to the **`definitions`** element of your endpoint's WSDL document.

Example 12.12. HTTP Provider WSDL Element's Namespace

```
<definitions ...
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"
```

Jetty runtime or Netty runtime

You can use the elements from the **`http-conf`** namespace to configure either the Jetty runtime or the Netty runtime.

The `server` element

The **`http-conf:server`** element is used to specify the connection properties of an HTTP service provider in a WSDL document. The **`http-conf:server`** element is a child of the WSDL **`port`** element. It has the same attributes as the **`server`** element used in the configuration file. The attributes are described in Table 12.5, "HTTP Service Provider Configuration Attributes".

Example

Example 12.13, "WSDL to Configure an HTTP Service Provider Endpoint" shows a WSDL fragment that configures an HTTP service provider endpoint specifying that it will not interact with caches.

Example 12.13. WSDL to Configure an HTTP Service Provider Endpoint

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:server CacheControl="no-cache" />
  </port>
</service>
```

12.3.3. Service Provider Cache Control Directives

Table 12.6, "`http-conf:server` Cache Control Directives" lists the cache control directives supported by an HTTP service provider.

Table 12.6. `http-conf:server` Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
public	Any cache can store the response.
private	Public (*shared*) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store any part of the response or any part of the request that invoked it.
no-transform	Caches must not modify the media type or location of the content in a response between a server and a client.
must-revalidate	Caches must revalidate expired entries that relate to a response before that entry can be used in a subsequent response.
proxy-revalidate	Does the same as must-revalidate, except that it can only be enforced on shared caches and is ignored by private unshared caches. When using this directive, the public cache directive must also be used.

Directive	Behavior
max-age	Clients can accept a response whose age is no greater that the specified number of seconds.
s-max-age	Does the same as max-age, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by s-max-age overrides the age specified by max-age. When using this directive, the proxy-revalidate directive must also be used.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

12.4. CONFIGURING THE JETTY RUNTIME

Overview

The Jetty runtime is used by HTTP service providers and HTTP consumers using a decoupled endpoint. The runtime's thread pool can be configured, and you can also set a number of the security settings for an HTTP service provider through the Jetty runtime.

Maven dependency

If you use Apache Maven as your build system, you can add the Jetty runtime to your project by including the following dependency in your project's **pom.xml** file:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-jetty</artifactId>
    <version>${cxf-version}</version>
</dependency>
```

Namespace

The elements used to configure the Jetty runtime are defined in the namespace http://cxf.apache.org/transports/http-jetty/configuration. It is commonly referred to using the prefix **httpj**. In order to use the Jetty configuration elements you must add the lines shown in Example 12.14, “Jetty Runtime Configuration Namespace” to the **beans** element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the **xsi:schemaLocation** attribute.

Example 12.14. Jetty Runtime Configuration Namespace

```
<beans ...
       xmlns:httpj="http://cxf.apache.org/transports/http-
```

```
jetty/configuration"
       ...
       xsi:schemaLocation="...
                           http://cxf.apache.org/transports/http-
jetty/configuration

http://cxf.apache.org/schemas/configuration/http-jetty.xsd
                          ...">
```

The `engine-factory` element

The **`httpj:engine-factory`** element is the root element used to configure the Jetty runtime used by an application. It has a single required attribute, **`bus`**, whose value is the name of the **`Bus`** that manages the Jetty instances being configured.

TIP

The value is typically **`cxf`** which is the name of the default **`Bus`** instance.

The **`httpj:engine-factory`** element has three children that contain the information used to configure the HTTP ports instantiated by the Jetty runtime factory. The children are described in Table 12.7, "Elements for Configuring a Jetty Runtime Factory".

Table 12.7. Elements for Configuring a Jetty Runtime Factory

Element	Description
`httpj:engine`	Specifies the configuration for a particular Jetty runtime instance. See the section called "The **`engine`** element".
`httpj:identifiedTLSServerParameters`	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, **`id`**, that specifies a unique identifier by which the property set can be referred.
`httpj:identifiedThreadingParameters`	Specifies a reusable set of properties for controlling a Jetty instance's thread pool. It has a single attribute, **`id`**, that specifies a unique identifier by which the property set can be referred. See the section called "Configuring the thread pool".

The `engine` element

The **`httpj:engine`** element is used to configure specific instances of the Jetty runtime. It has a single attribute, **`port`**, that specifies the number of the port being managed by the Jetty instance.

TIP

You can specify a value of `0` for the `port` attribute. Any threading properties specified in an `httpj:engine` element with its `port` attribute set to `0` are used as the configuration for all Jetty listeners that are not explicitly configured.

Each `httpj:engine` element can have two children: one for configuring security properties and one for configuring the Jetty instance's thread pool. For each type of configuration you can either directly provide the configuration information or you can provide a reference to a set of configuration properties defined in the parent `httpj:engine-factory` element.

The child elements used to provide the configuration properties are described in Table 12.8, "Elements for Configuring a Jetty Runtime Instance".

Table 12.8. Elements for Configuring a Jetty Runtime Instance

Element	Description
`httpj:tlsServerParameters`	Specifies a set of properties for configuring the security used for the specific Jetty instance.
`httpj:tlsServerParametersRef`	Refers to a set of security properties defined by a `identifiedTLSServerParameters` element. The `id` attribute provides the id of the referred `identifiedTLSServerParameters` element.
`httpj:threadingParameters`	Specifies the size of the thread pool used by the specific Jetty instance. See the section called "Configuring the thread pool".
`httpj:threadingParametersRef`	Refers to a set of properties defined by a `identifiedThreadingParameters` element. The `id` attribute provides the id of the referred `identifiedThreadingParameters` element.

Configuring the thread pool

You can configure the size of a Jetty instance's thread pool by either:

- Specifying the size of the thread pool using a `identifiedThreadingParameters` element in the `engine-factory` element. You then refer to the element using a `threadingParametersRef` element.
- Specifying the size of the of the thread pool directly using a `threadingParameters` element.

The `threadingParameters` has two attributes to specify the size of a thread pool. The attributes are described in Table 12.9, "Attributes for Configuring a Jetty Thread Pool" .

NOTE

The `httpj:identifiedThreadingParameters` element has a single child `threadingParameters` element.

Table 12.9. Attributes for Configuring a Jetty Thread Pool

Attribute	Description
`minThreads`	Specifies the minimum number of threads available to the Jetty instance for processing requests.
`maxThreads`	Specifies the maximum number of threads available to the Jetty instance for processing requests.

Example

Example 12.15, "Configuring a Jetty Instance" shows a configuration fragment that configures a Jetty instance on port number 9001.

Example 12.15. Configuring a Jetty Instance

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
  xsi:schemaLocation="http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
            http://cxf.apache.org/transports/http/configuration
            http://cxf.apache.org/schemas/configuration/http-conf.xsd
            http://cxf.apache.org/transports/http-jetty/configuration
            http://cxf.apache.org/schemas/configuration/http-jetty.xsd
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd">
  ...

  <httpj:engine-factory bus="cxf">
    <httpj:identifiedTLSServerParameters id="secure">
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="JKS" password="password"
                      file="certs/cherry.jks"/>
      </sec:keyManagers>
    </httpj:identifiedTLSServerParameters>

    <httpj:engine port="9001">
      <httpj:tlsServerParametersRef id="secure" />
      <httpj:threadingParameters minThreads="5"
                                 maxThreads="15" />
    </httpj:engine>
  </httpj:engine-factory>
 </beans>
```

12.5. CONFIGURING THE NETTY RUNTIME

Overview

The Netty runtime is used by HTTP service providers and HTTP consumers using a decoupled endpoint. The runtime's thread pool can be configured, and you can also set a number of the security settings for an HTTP service provider through the Netty runtime.

Maven dependencies

If you use Apache Maven as your build system, you can add the server-side implementation of the Netty runtime (for defining Web service endpoints) to your project by including the following dependency in your project's **pom.xml** file:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-netty-server</artifactId>
    <version>${cxf-version}</version>
</dependency>
```

You can add the client-side implementation of the Netty runtime (for defining Web service clients) to your project by including the following dependency in your project's **pom.xml** file:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-netty-client</artifactId>
    <version>${cxf-version}</version>
</dependency>
```

Namespace

The elements used to configure the Netty runtime are defined in the namespace http://cxf.apache.org/transports/http-netty-server/configuration. It is commonly referred to using the prefix **httpn**. In order to use the Netty configuration elements you must add the lines shown in Example 12.16, "Netty Runtime Configuration Namespace" to the **beans** element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the **xsi:schemaLocation** attribute.

Example 12.16. Netty Runtime Configuration Namespace

```
<beans ...
       xmlns:httpn="http://cxf.apache.org/transports/http-netty-
server/configuration"
       ...
       xsi:schemaLocation="...
              http://cxf.apache.org/transports/http-netty-
server/configuration
           http://cxf.apache.org/schemas/configuration/http-netty-
server.xsd
              ...">
```

The engine-factory element

The **httpn:engine-factory** element is the root element used to configure the Netty runtime used

by an application. It has a single required attribute, **bus**, whose value is the name of the **Bus** that manages the Netty instances being configured.

TIP

The value is typically **cxf**, which is the name of the default **Bus** instance.

The **httpn:engine-factory** element has three children that contain the information used to configure the HTTP ports instantiated by the Netty runtime factory. The children are described in Table 12.10, "Elements for Configuring a Netty Runtime Factory" .

Table 12.10. Elements for Configuring a Netty Runtime Factory

Element	Description
httpn:engine	Specifies the configuration for a particular Netty runtime instance. See the section called "The **engine** element".
httpn:identifiedTLSServerParameters	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, **id**, that specifies a unique identifier by which the property set can be referred.
httpn:identifiedThreadingParameters	Specifies a reusable set of properties for controlling a Netty instance's thread pool. It has a single attribute, **id**, that specifies a unique identifier by which the property set can be referred. See the section called "Configuring the thread pool".

The engine element

The **httpn:engine** element is used to configure specific instances of the Netty runtime. Table 12.11, "Attributes for Configuring a Netty Runtime Instance" shows the attributes supported by the **httpn:engine** element.

Table 12.11. Attributes for Configuring a Netty Runtime Instance

Attribute	Description
port	Specifies the port used by the Netty HTTP server instance. You can specify a value of **0** for the port attribute. Any threading properties specified in an engine element with its port attribute set to **0** are used as the configuration for all Netty listeners that are not explicitly configured.
host	Specifies the listen address used by the Netty HTTP server instance. The value can be a hostname or an IP address. If not specified, Netty HTTP server will listen on all local addresses.

Attribute	Description
readIdleTime	Specifies the maximum read idle time for a Netty connection. The timer is reset whenever there are any read actions on the underlying stream.
writeIdleTime	Specifies the maximum write idle time for a Netty connection. The timer is reset whenever there are any write actions on the underlying stream.
maxChunkContentSize	Specifies the maximum aggregated content size for a Netty connection. The default value is 10MB.

A **httpn:engine** element has one child element for configuring security properties and one child element for configuring the Netty instance's thread pool. For each type of configuration you can either directly provide the configuration information or you can provide a reference to a set of configuration properties defined in the parent **httpn:engine-factory** element.

The supported child elements of **httpn:engine** are shown in Table 12.12, "Elements for Configuring a Netty Runtime Instance".

Table 12.12. Elements for Configuring a Netty Runtime Instance

Element	Description
httpn:tlsServerParameters	Specifies a set of properties for configuring the security used for the specific Netty instance.
httpn:tlsServerParametersRef	Refers to a set of security properties defined by a **identifiedTLSServerParameters** element. The **id** attribute provides the id of the referred **identifiedTLSServerParameters** element.
httpn:threadingParameters	Specifies the size of the thread pool used by the specific Netty instance. See the section called "Configuring the thread pool".
httpn:threadingParametersRef	Refers to a set of properties defined by a **identifiedThreadingParameters** element. The **id** attribute provides the id of the referred **identifiedThreadingParameters** element.
httpn:sessionSupport	When **true**, enables support for HTTP sessions. Default is **false**.
httpn:reuseAddress	Specifies a boolean value to set the **ReuseAddress** TCP socket option. Default is **false**.

Configuring the thread pool

You can configure the size of a Netty instance's thread pool by either:

- Specifying the size of the thread pool using a **identifiedThreadingParameters** element in the **engine-factory** element. You then refer to the element using a **threadingParametersRef** element.
- Specifying the size of the of the thread pool directly using a **threadingParameters** element.

The **threadingParameters** element has one attribute to specify the size of a thread pool, as described in Table 12.13, “Attributes for Configuring a Netty Thread Pool” .

NOTE

The **httpn:identifiedThreadingParameters** element has a single child **threadingParameters** element.

Table 12.13. Attributes for Configuring a Netty Thread Pool

Attribute	Description
threadPoolSize	Specifies the number of threads available to the Netty instance for processing requests.

Example

Example 12.17, “Configuring a Netty Instance” shows a configuration fragment that configures a variety of Netty ports.

Example 12.17. Configuring a Netty Instance

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:h="http://cxf.apache.org/transports/http/configuration"
       xmlns:httpn="http://cxf.apache.org/transports/http-netty-
server/configuration"
       xmlns:sec="http://cxf.apache.org/configuration/security"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-
beans.xsd
        http://cxf.apache.org/configuration/security
            http://cxf.apache.org/schemas/configuration/security.xsd
        http://cxf.apache.org/transports/http/configuration
            http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://cxf.apache.org/transports/http-netty-
server/configuration
            http://cxf.apache.org/schemas/configuration/http-netty-
server.xsd"
>
    ...
    <httpn:engine-factory bus="cxf">
```

```
        <httpn:identifiedTLSServerParameters id="sample1">
          <httpn:tlsServerParameters jsseProvider="SUN"
secureSocketProtocol="TLS">
            <sec:clientAuthentication want="false" required="false"/>
          </httpn:tlsServerParameters>
        </httpn:identifiedTLSServerParameters>

        <httpn:identifiedThreadingParameters id="sampleThreading1">
           <httpn:threadingParameters threadPoolSize="120"/>
        </httpn:identifiedThreadingParameters>

        <httpn:engine port="9000" readIdleTime="30000"
writeIdleTime="90000">
           <httpn:threadingParametersRef id="sampleThreading1"/>
        </httpn:engine>

        <httpn:engine port="0">
           <httpn:threadingParameters threadPoolSize="400"/>
        </httpn:engine>

        <httpn:engine port="9001" readIdleTime="40000"
maxChunkContentSize="10000">
          <httpn:threadingParameters threadPoolSize="99" />
          <httpn:sessionSupport>true</httpn:sessionSupport>
        </httpn:engine>

        <httpn:engine port="9002">
          <httpn:tlsServerParameters>
            <sec:clientAuthentication want="true" required="true"/>
          </httpn:tlsServerParameters>
        </httpn:engine>

        <httpn:engine port="9003">
           <httpn:tlsServerParametersRef id="sample1"/>
        </httpn:engine>

    </httpn:engine-factory>
</beans>
```

12.6. USING THE HTTP TRANSPORT IN DECOUPLED MODE

Overview

In normal HTTP request/response scenarios, the request and the response are sent using the same HTTP connection. The service provider processes the request and responds with a response containing the appropriate HTTP status code and the contents of the response. In the case of a successful request, the HTTP status code is set to **200**.

In some instances, such as when using WS-RM or when requests take an extended period of time to execute, it makes sense to decouple the request and response message. In this case the service providers sends the consumer a **202 Accepted** response to the consumer over the back-channel of the HTTP connection on which the request was received. It then processes the request and sends the

response back to the consumer using a new decoupled server->client HTTP connection. The consumer runtime receives the incoming response and correlates it with the appropriate request before returning to the application code.

Configuring decoupled interactions

Using the HTTP transport in decoupled mode requires that you do the following:

1. Configure the consumer to use WS-Addressing.

   See the section called “Configuring an endpoint to use WS-Addressing” .

2. Configure the consumer to use a decoupled endpoint.

   See the section called “Configuring the consumer”.

3. Configure any service providers that the consumer interacts with to use WS-Addressing.

   See the section called “Configuring an endpoint to use WS-Addressing” .

Configuring an endpoint to use WS-Addressing

Specify that the consumer and any service provider with which the consumer interacts use WS-Addressing.

You can specify that an endpoint uses WS-Addressing in one of two ways:

- Adding the **wswa:UsingAddressing** element to the endpoint's WSDL **port** element as shown in Example 12.18, “Activating WS-Addressing using WSDL” .

  Example 12.18. Activating WS-Addressing using WSDL

  ```
  ...
  <service name="WidgetSOAPService">
    <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
      <soap:address="http://widgetvendor.net/widgetSeller" />
      <wswa:UsingAddressing
  xmlns:wswa="http://www.w3.org/2005/02/addressing/wsdl"/>
    </port>
  </service>
  ...
  ```

- Adding the WS-Addressing policy to the endpoint's WSDL **port** element as shown in Example 12.19, “Activating WS-Addressing using a Policy”.

  Example 12.19. Activating WS-Addressing using a Policy

  ```
  ...
  <service name="WidgetSOAPService">
    <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
      <soap:address="http://widgetvendor.net/widgetSeller" />
      <wsp:Policy xmlns:wsp="http://www.w3.org/2006/07/ws-policy">
        <wsam:Addressing
  xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
  ```

```
        <wsp:Policy/>
      </wsam:Addressing>
    </wsp:Policy>
  </port>
</service>
...
```

NOTE

The WS-Addressing policy supersedes the **wswa:UsingAddressing** WSDL element.

Configuring the consumer

Configure the consumer endpoint to use a decoupled endpoint using the **DecoupledEndpoint** attribute of the **http-conf:conduit** element.

Example 12.20, "Configuring a Consumer to Use a Decoupled HTTP Endpoint" shows the configuration for setting up the endpoint defined in Example 12.18, "Activating WS-Addressing using WSDL" to use use a decoupled endpoint. The consumer now receives all responses at http://widgetvendor.net/widgetSellerInbox.

Example 12.20. Configuring a Consumer to Use a Decoupled HTTP Endpoint

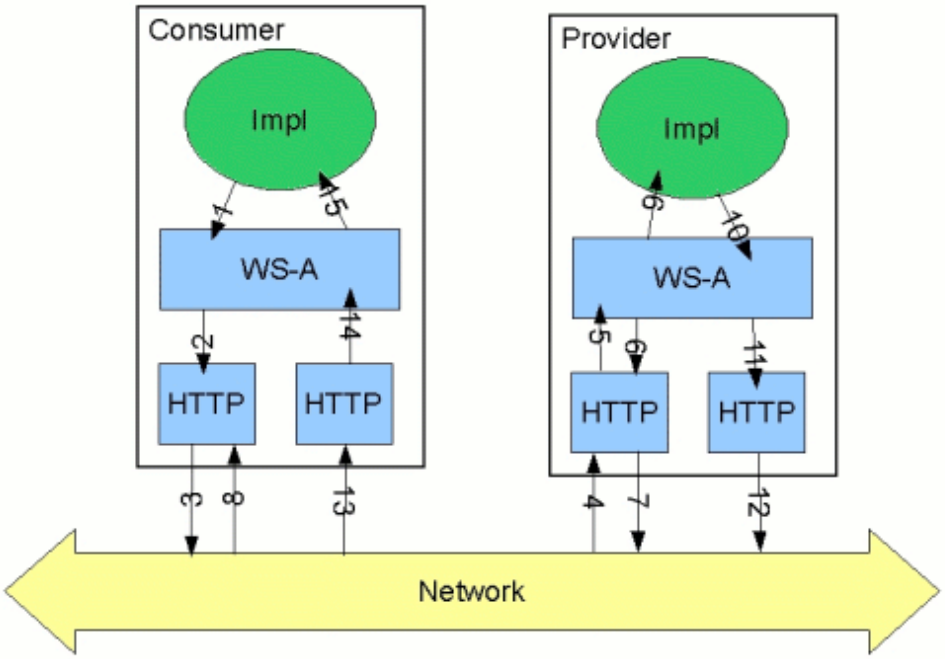
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"

xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration

http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http:conduit name="
{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">
    <http:client
DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
  </http:conduit>
</beans>
```

How messages are processed

Using the HTTP transport in decoupled mode adds extra layers of complexity to the processing of HTTP messages. While the added complexity is transparent to the implementation level code in an application, it might be important to understand what happens for debugging reasons.

Figure 12.1, "Message Flow in for a Decoupled HTTP Transport" shows the flow of messages when using HTTP in decoupled mode.

Figure 12.1. Message Flow in for a Decoupled HTTP Transport



A request starts the following process:

1. The consumer implementation invokes an operation and a request message is generated.
2. The WS-Addressing layer adds the WS-A headers to the message.

   When a decoupled endpoint is specified in the consumer's configuration, the address of the decoupled endpoint is placed in the WS-A ReplyTo header.
3. The message is sent to the service provider.
4. The service provider receives the message.
5. The request message from the consumer is dispatched to the provider's WS-A layer.
6. Because the WS-A ReplyTo header is not set to anonymous, the provider sends back a message with the HTTP status code set to **202**, acknowledging that the request has been received.
7. The HTTP layer sends a **202 Accepted** message back to the consumer using the original connection's back-channel.
8. The consumer receives the **202 Accepted** reply on the back-channel of the HTTP connection used to send the original message.

   When the consumer receives the **202 Accepted** reply, the HTTP connection closes.

9. The request is passed to the service provider's implementation where the request is processed.
10. When the response is ready, it is dispatched to the WS-A layer.
11. The WS-A layer adds the WS-Addressing headers to the response message.
12. The HTTP transport sends the response to the consumer's decoupled endpoint.
13. The consumer's decoupled endpoint receives the response from the service provider.
14. The response is dispatched to the consumer's WS-A layer where it is correlated to the proper request using the WS-A RelatesTo header.
15. The correlated response is returned to the client implementation and the invoking call is unblocked.

CHAPTER 13. USING SOAP OVER JMS

Abstract

Apache CXF implements the W3C standard SOAP/JMS transport. This standard is intended to provide a more robust alternative to SOAP/HTTP services. Apache CXF applications using this transport should be able to interoperate with applications that also implement the SOAP/JMS standard. The transport is configured directly in an endpoint's WSDL.

13.1. BASIC CONFIGURATION

Overview

The SOAP over JMS protocol is defined by the World Wide Web Consortium(W3C) as a way of providing a more reliable transport layer to the customary SOAP/HTTP protocol used by most services. The Apache CXF implementation is fully compliant with the specification and should be compatible with any framework that is also compliant.

This transport uses JNDI to find the JMS destinations. When an operation is invoked, the request is packaged as a SOAP message and sent in the body of a JMS message to the specified destination.

To use the SOAP/JMS transport:

1. Specify that the transport type is SOAP/JMS.
2. Specify the target destination using a JMS URI.
3. Optionally, configure the JNDI connection.
4. Optionally, add additional JMS configuration.

Specifying the JMS transport type

You configure a SOAP binding to use the JMS transport when specifying the WSDL binding. You set the **soap:binding** element's **transport** attribute to **http://www.w3.org/2010/soapjms/**. Example 13.1, “SOAP over JMS binding specification” shows a WSDL binding that uses SOAP/JMS.

Example 13.1. SOAP over JMS binding specification

```
<wsdl:binding ... >
  <soap:binding style="document"
                transport="http://www.w3.org/2010/soapjms/" />
  ...
</wsdl:binding>
```

Specifying the target destination

You specify the address of the JMS target destination when specifying the WSDL port for the endpoint. The address specification for a SOAP/JMS endpoint uses the same **soap:address** element and attribute as a SOAP/HTTP endpoint. The difference is the address specification. JMS endpoints use a

JMS URI as defined in the URI Scheme for JMS 1.0 . Example 13.2, "JMS URI syntax" shows the syntax for a JMS URI.

Example 13.2. JMS URI syntax

```
jms:variant:destination?options
```

Table 13.1, "JMS URI variants" describes the available variants for the JMS URI.

Table 13.1. JMS URI variants

Variant	Description
jndi	Specifies that the destination is a JNDI name for the target destination. When using this variant, you must provide the configuration for accessing the JNDI provider.
topic	Specifies that the destination is the name of the topic to be used as the target destination. The string provided is passed into **`Session.createTopic()`** to create a representation of the destination.
queue	Specifies that the destination is the name of the queue to be used as the target destination. The string provided is passed into **`Session.createQueue()`** to create a representation of the destination.

The *options* portion of a JMS URI are used to configure the transport and are discussed in Section 13.2, "JMS URIs".

Example 13.3, "SOAP/JMS endpoint address" shows the WSDL port entry for a SOAP/JMS endpoint whose target destination is looked up using JNDI.

Example 13.3. SOAP/JMS endpoint address

```
<wsdl:port ... >
  ...
  <soap:address
location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
</wsdl:port>
```

For working with SOAP/JMS services in Java see Chapter 39, *Using SOAP over JMS*.

Configuring JNDI and the JMS transport

The SOAP/JMS provides several ways to configure the JNDI connection and the JMS transport:

- Using the JMS URI
- Using WSDL extensions

13.2. JMS URIS

Overview

When using SOAP/JMS, a JMS URI is used to specify the endpoint's target destination. The JMS URI can also be used to configure JMS connection by appending one or more options to the URI. These options are detailed in the IETF standard, URI Scheme for Java Message Service 1.0 . They can be used to configure the JNDI system, the reply destination, the delivery mode to use, and other JMS properties.

Syntax

As shown in Example 13.2, "JMS URI syntax", you can append one or more options to the end of a JMS URI by separating them from the destination's address with a question mark(**?**). Multiple options are separated by an ampersand(**&**). Example 13.4, "Syntax for JMS URI options" shows the syntax for using multiple options in a JMS URI.

Example 13.4. Syntax for JMS URI options

```
jmsAddress?option1=value1&option2=value2&...optionN=valueN
```

JMS properties

Table 13.2, "JMS properties settable as URI options" shows the URI options that affect the JMS transport layer.

Table 13.2. JMS properties settable as URI options

Property	Default	Description
`deliveryMode`	**PERSISTENT**	Specifies whether to use JMS **PERSISTENT** or **NON_PERSISTENT** message semantics. In the case of **PERSISTENT** delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas **NON_PERSISTENT** messages are kept in memory only.

Property	Default	Description
`replyToName`		Explicitly specifies the reply destination to appear in the **`JMSReplyTo`** header. Setting this property is recommended for applications that have request-reply semantics because the JMS provider will assign a temporary reply queue if one is not explicitly set. The value of this property has an interpretation that depends on the variant specified in the JMS URI: • **`jndi`** variant—the JNDI name of the destination • **`queue`** or **`topic`** variants—the actual name of the destination
`priority`	`4`	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`timeToLive`	`0`	Time (in milliseconds) after which the message will be discarded by the JMS provider. A value of **`0`** represents an infinite lifetime (the default).

JNDI properties

Table 13.3, "JNDI properties settable as URI options" shows the URI options that can be used to configure JNDI for this endpoint.

Table 13.3. JNDI properties settable as URI options

Property	Description
`jndiConnectionFactoryName`	Specifies the JNDI name of the JMS connection factory.
`jndiInitialContextFactory`	Specifies the fully qualified Java class name of the JNDI provider (which must be of **`javax.jms.InitialContextFactory`** type). Equivalent to setting the **`java.naming.factory.initial`** Java system property.

Property	Description
jndiURL	Specifies the URL that initializes the JNDI provider. Equivalent to setting the **java.naming.provider.url** Java system property.

Additional JNDI properties

The properties, **java.naming.factory.initial** and **java.naming.provider.url**, are standard properties, which are required to initialize any JNDI provider. Sometimes, however, a JNDI provider might support custom properties in addition to the standard ones. In this case, you can set an arbitrary JNDI property by setting a URI option of the form **jndi-*PropertyName***.

For example, if you were using SUN's LDAP implementation of JNDI, you could set the JNDI property, **java.naming.factory.control**, in a JMS URI as shown in Example 13.5, "Setting a JNDI property in a JMS URI".

Example 13.5. Setting a JNDI property in a JMS URI

```
jms:queue:FOO.BAR?jndi-
java.naming.factory.control=com.sun.jndi.ldap.ResponseControlFactory
```

Example

If the JMS provider is *not* already configured, it is possible to provide the requisite JNDI configuration details in the URI using options (see Table 13.3, "JNDI properties settable as URI options"). For example, to configure an endpoint to use the Apache ActiveMQ JMS provider and connect to the queue called **test.cxf.jmstransport.queue**, use the URI shown in Example 13.6, "JMS URI that configures a JNDI connection".

Example 13.6. JMS URI that configures a JNDI connection

```
jms:jndi:dynamicQueues/test.cxf.jmstransport.queue
?
jndiInitialContextFactory=org.apache.activemq.jndi.ActiveMQInitialContex
tFactory
&jndiConnectionFactoryName=ConnectionFactory
&jndiURL=tcp://localhost:61616
```

13.3. WSDL EXTENSIONS

Overview

You can specify the basic configuration of the JMS transport by inserting WSDL extension elements into the contract, either at binding scope, service scope, or port scope. The WSDL extensions enable you to specify the properties for bootstrapping a JNDI **InitialContext**, which can then be used to

look up JMS destinations. You can also set some properties that affect the behavior of the JMS transport layer.

SOAP/JMS namespace

the SOAP/JMS WSDL extensions are defined in the **`http://www.w3.org/2010/soapjms/`** namespace. To use them in your WSDL contracts add the following setting to the **`wsdl:definitions`** element:

```
<wsdl:definitions ...
    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
```

WSDL extension elements

Table 13.4, "SOAP/JMS WSDL extension elements" shows all of the WSDL extension elements you can use to configure the JMS transport.

Table 13.4. SOAP/JMS WSDL extension elements

Element	Default	Description
`soapjms:jndiInitialContextFactory`		Specifies the fully qualified Java class name of the JNDI provider. Equivalent to setting the **`java.naming.factory.initial`** Java system property.
`soapjms:jndiURL`		Specifies the URL that initializes the JNDI provider. Equivalent to setting the **`java.naming.provider.url`** Java system property.
`soapjms:jndiContextParameter`		Enables you to specify an additional property for creating the JNDI **`InitialContext`**. Use the **`name`** and **`value`** attributes to specify the property.
`soapjms:jndiConnectionFactoryName`		Specifies the JNDI name of the JMS connection factory.

Element	Default	Description
`soapjms:deliveryMode`	**PERSISTENT**	Specifies whether to use JMS **PERSISTENT** or **NON_PERSISTENT** message semantics. In the case of **PERSISTENT** delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas **NON_PERSISTENT** messages are kept in memory only.
`soapjms:replyToName`		Explicitly specifies the reply destination to appear in the **JMSReplyTo** header. Setting this property is recommended for SOAP invocations that have request-reply semantics. If this property is not set the JMS provider allocates a temporary queue with an automatically generated name. The value of this property has an interpretation that depends on the variant specified in the JMS URI, as follows: • **jndi** variant—the JNDI name of the destination. • **queue** or **topic** variants—the actual name of the destination.
`soapjms:priority`	**4**	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`soapjms:timeToLive`	**0**	Time, in milliseconds, after which the message will be discarded by the JMS provider. A value of **0** represents an infinite lifetime.

Configuration scopes

The WSDL elements placement in the WSDL contract effect the scope of the configuration changes on the endpoints defined in the contract. The SOAP/JMS WSDL elements can be placed as children of either the `wsdl:binding` element, the `wsdl:service` element, or the `wsdl:port` element. The parent of the SOAP/JMS elements determine which of the following scopes the configuration is placed into.

Binding scope

You can configure the JMS transport at the *binding scope* by placing extension elements inside the **wsdl:binding** element. Elements in this scope define the default configuration for all endpoints that use this binding. Any settings in the binding scope can be overridden at the service scope or the port scope.

Service scope

You can configure the JMS transport at the *service scope* by placing extension elements inside a **wsdl:service** element. Elements in this scope define the default configuration for all endpoints in this service. Any settings in the service scope can be overridden at the port scope.

Port scope

You can configure the JMS transport at the *port scope* by placing extension elements inside a **wsdl:port** element. Elements in the port scope define the configuration for this port. They override any defaults defined at the service scope or at the binding scope.

Example

Example 13.7, "WSDL contract with SOAP/JMS configuration" shows a WSDL contract for a SOAP/JMS service. It configures the JNDI layer in the binding scope, the message delivery details in the service scope, and the reply destination in the port scope.

Example 13.7. WSDL contract with SOAP/JMS configuration

```
<wsd;definitions ...
1     xmlns:soapjms="http://www.w3.org/2010/soapjms/"
    ... >
  ...
  <wsdl:binding name="JMSGreeterPortBinding"
type="tns:JMSGreeterPortType">
    ...
2     <soapjms:jndiInitialContextFactory>
        org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </soapjms:jndiInitialContextFactory>
    <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
    <soapjms:jndiConnectionFactoryName>
      ConnectionFactory
    </soapjms:jndiConnectionFactoryName>
    ...
  </wsdl:binding>
  ...
  <wsdl:service name="JMSGreeterService">
    ...
3     <soapjms:deliveryMode>NON_PERSISTENT</soapjms:deliveryMode>
      <soapjms:timeToLive>60000</soapjms:timeToLive>
    ...
    <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
4       <soap:address
  location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
5       <soapjms:replyToName>
          dynamicQueues/greeterReply.queue
      </soapjms:replyToName>
      ...
    </wsdl:port>
```

```
    ...
  </wsdl:service>
  ...
</wsdl:definitions>
```

The WSDL in Example 13.7, "WSDL contract with SOAP/JMS configuration" does the following:

1. Declare the namespace for the SOAP/JMS extensions.
2. Configure the JNDI connections in the binding scope.
3. Configure the JMS delivery style to non-persistent and each message to live for one minute.
4. Specify the target destination.
5. Configure the JMS transport so that reply messages are delivered on the **greeterReply.queue** queue.

CHAPTER 14. USING GENERIC JMS

Abstract

Apache CXF provides a generic implementation of a JMS transport. The generic JMS transport is not restricted to using SOAP messages and allows for connecting to any application that uses JMS.

The Apache CXF generic JMS transport can connect to any JMS provider and work with applications that exchange JMS messages with bodies of either **`TextMessage`** or **`ByteMessage`**.

There are two ways to enable and configure the JMS transport:

- JMS configuration bean
- WSDL

14.1. USING THE JMS CONFIGURATION BEAN

Overview

To simplify JMS configuration and make it more powerful, Apache CXF uses a single JMS configuration bean to configure JMS endpoints. The bean is implemented by the **`org.apache.cxf.transport.jms.JMSConfiguration`** class. It can be used to either configure endpoint's directly or to configure the JMS conduits and destinations.

Configuration namespace

The JMS configuration bean uses the Spring p-namespace to make the configuration as simple as possible. To use this namespace you need to declare it in the configuration's root element as shown in Example 14.1, “Declaring the Spring p-namespace”.

Example 14.1. Declaring the Spring p-namespace

```
<beans ...
  xmlns:p="http://www.springframework.org/schema/p"
  ... >
  ...
</beans>
```

Specifying the configuration

You specify the JMS configuration by defining a bean of class **`org.apache.cxf.transport.jms.JMSConfiguration`**. The properties of the bean provide the configuration settings for the transport.

Table 14.1, “General JMS Configuration Properties” lists properties that are common to both providers and consumers.

Table 14.1. General JMS Configuration Properties

Property	Default	Description
connectionFactory-ref		Specifies a reference to a bean that defines a JMS **ConnectionFactory**.
wrapInSingleConnectionFactory	**true**	Specifies whether to wrap the **ConnectionFactory** with a Spring **SingleConnectionFactory**. Doing so can improve the performance of the JMS transport when the specified connection factory does not pool connections.
reconnectOnException	**false**	Specifies whether to create a new connection in the case of an exception. This property is only used when wrapping the connection factory with a Spring **SingleConnectionFactory**.
targetDestination		Specifies the JNDI name or provider specific name of a destination.
replyDestination		Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 14.3, "Using a Named Reply Destination".
destinationResolver		Specifies a reference to a Spring **DestinationResolver**. This allows you to define how destination names are resolved. By default a **DynamicDestinationResolver** is used. It resolves destinations using the JMS providers features. If you reference a **JndiDestinationResolver** you can resolve the destination names using JNDI.

Property	Default	Description
transactionManager		Specifies a reference to a Spring transaction manager. This allows the service to participate in JTA Transactions.
taskExecutor		Specifies a reference to a Spring **TaskExecutor**. This is used in listeners to decide how to handle incoming messages. By default the transport uses the Spring **SimpleAsyncTaskExecutor**.
useJms11	**false**	Specifies whether JMS 1.1 features are available.
messageIdEnabled	**true**	Specifies whether the JMS transport wants the JMS broker to provide message IDs. Setting this to **false** causes the endpoint to call its message producer's **setDisableMessageID()** method with a value of **true**. The JMS broker is then given a hint that it does not need to generate message IDs or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.
messageTimestampEnabled	**true**	Specifies whether the JMS transport wants the JMS broker to provide message time stamps. Setting this to **false** causes the endpoint to call its message producer's **setDisableMessageTimestamp()** method with a value of **true**. The JMS broker is then given a hint that it does not need to generate time stamps or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.

Property	Default	Description
cacheLevel	**3**	Specifies the level of caching allowed by the listener. Valid values are **0**(CACHE_NONE), **1**(CACHE_CONNECTION), **2**(CACHE_SESSION), **3**(CACHE_CONSUMER), **4**(CACHE_AUTO).
pubSubNoLocal	**false**	Specifies whether to receive messages produced from the same connection.
receiveTimeout	**0**	Specifies, in milliseconds, the amount of time to wait for response messages. **0** means wait indefinitely.
explicitQosEnabled	**false**	Specifies whether the QoS settings like priority, persistence, and time to live are explicitly set for each message or if they are allowed to use default values.
deliveryMode	**1**	Specifies if a message is persistent. The two values are: • **1**(NON_PERSISTENT)—messages will be kept memory • **2**(PERSISTENT)—messages will be persisted to disk
priority	**4**	Specifies the message's priority for the messages. JMS priority values can range from 0 to 9. The lowest priority is 0 and the highest priority is 9.
timeToLive	**0**	Specifies, in milliseconds, the message will be available after it is sent. 0 specifies an infinite time to live.
sessionTransacted	**false**	Specifies if JMS transactions are used.

Property	Default	Description
concurrentConsumers	**1**	Specifies the minimum number of concurrent consumers created by the listener.
maxConcurrentConsumers	**1**	Specifies the maximum number of concurrent consumers by listener.
messageSelector		Specifies the string value of the selector. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
subscriptionDurable	**false**	Specifies whether the server uses durrable subscriptions.
durableSubscriptionName		Specifies the string used to register the durable subscription.
messageType	**text**	Specifies how the message data will be packaged as a JMS message. **text** specifies that the data will be packaged as a **TextMessage**. **binary** specifies that the data will be packaged as an **ByteMessage**.
pubSubDomain	**false**	Specifies whether the target destination is a topic.
jmsProviderTibcoEms	**false**	Specifies if your JMS provider is Tibco EMS. This causes the principal in the security context to be populated from the **JMS_TIBCO_SENDER** header.
useMessageIDAsCorrelationID	**false**	Specifies whether JMS will use the message ID to correlate messages. If not, the client will set a generated correlation ID.

As shown in Example 14.2, "JMS configuration bean", the bean's properties are specified as attributes to the **bean** element. They are all declared in the Spring **p** namespace.

Example 14.2. JMS configuration bean

```
<bean id="jmsConfig"
      class="org.apache.cxf.transport.jms.JMSConfiguration"
```

```
        p:connectionFactory-ref="connectionFactory"
        p:targetDestination="dynamicQueues/greeter.request.queue"
        p:pubSubDomain="false" />
```

Applying the configuration to an endpoint

The **JMSConfiguration** bean can be applied directly to both server and client endpoints using the Apache CXF features mechanism. To do so:

1. Set the endpoint's **address** attribute to **jms://**.
2. Add a **jaxws:feature** element to the endpoint's configuration.
3. Add a bean of type **org.apache.cxf.transport.jms.JMSConfigFeature** to the feature.
4. Set the **bean** element's **p:jmsConfig-ref** attribute to the ID of the **JMSConfiguration** bean.

Example 14.3, “Adding JMS configuration to a JAX-WS client” shows a JAX-WS client that uses the JMS configuration from Example 14.2, “JMS configuration bean”.

Example 14.3. Adding JMS configuration to a JAX-WS client

```
<jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              serviceName="customer:CustomerServiceService"
              endpointName="customer:CustomerServiceEndpoint"
              address="jms://"

serviceClass="com.example.customerservice.CustomerService">
  <jaxws:features>
    <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
          p:jmsConfig-ref="jmsConfig"/>
  </jaxws:features>
</jaxws:client>
```

Applying the configuration to the transport

The **JMSConfiguration** bean can be applied to JMS conduits and JMS destinations using the **jms:jmsConfig-ref** element. The **jms:jmsConfig-ref** element's value is the ID of the **JMSConfiguration** bean.

Example 14.4, “Adding JMS configuration to a JMS conduit” shows a JMS conduit that uses the JMS configuration from Example 14.2, “JMS configuration bean”.

Example 14.4. Adding JMS configuration to a JMS conduit

```
<jms:conduit name="
{http://cxf.apache.org/jms_conf_test}HelloWorldQueueBinMsgPort.jms-
conduit">
```

```
    ...
    <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

14.2. USING WSDL TO CONFIGURE JMS

The WSDL extensions for defining a JMS endpoint are defined in the namespace http://cxf.apache.org/transports/jms. In order to use the JMS extensions you will need to add the line shown in Example 14.5, "JMS WSDL extension namespace" to the definitions element of your contract.

Example 14.5. JMS WSDL extension namespace

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

14.2.1. Basic JMS configuration

Overview

The JMS address information is provided using the **jms:address** element and its child, the **jms:JMSNamingProperties** element. The **jms:address** element's attributes specify the information needed to identify the JMS broker and the destination. The **jms:JMSNamingProperties** element specifies the Java properties used to connect to the JNDI service.

IMPORTANT

Information specified using the JMS feature will override the information in the endpoint's WSDL file.

Specifying the JMS address

The basic configuration for a JMS endpoint is done by using a **jms:address** element as the child of your service's **port** element. The **jms:address** element used in WSDL is identical to the one used in the configuration file. Its attributes are listed in Table 14.2, "JMS endpoint attributes".

Table 14.2. JMS endpoint attributes

Attribute	Description
destinationStyle	Specifies if the JMS destination is a JMS queue or a JMS topic.
jndiConnectionFactoryName	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
jmsDestinationName	Specifies the JMS name of the JMS destination to which requests are sent.

Attribute	Description
jmsReplyDestinationName	Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 14.3, "Using a Named Reply Destination".
jndiDestinationName	Specifies the JNDI name bound to the JMS destination to which requests are sent.
jndiReplyDestinationName	Specifies the JNDI name bound to the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 14.3, "Using a Named Reply Destination".
connectionUserName	Specifies the user name to use when connecting to a JMS broker.
connectionPassword	Specifies the password to use when connecting to a JMS broker.

The **jms:address** WSDL element uses a **jms:JMSNamingProperties** child element to specify additional information needed to connect to a JNDI provider.

Specifying JNDI properties

To increase interoperability with JMS and JNDI providers, the **jms:address** element has a child element, **jms:JMSNamingProperties**, that allows you to specify the values used to populate the properties used when connecting to the JNDI provider. The **jms:JMSNamingProperties** element has two attributes: **name** and **value**. **name** specifies the name of the property to set. **value** attribute specifies the value for the specified property. **jms:JMSNamingProperties** element can also be used for specification of provider specific properties.

The following is a list of common JNDI properties that can be set:

1. **java.naming.factory.initial**
2. **java.naming.provider.url**
3. **java.naming.factory.object**
4. **java.naming.factory.state**
5. **java.naming.factory.url.pkgs**
6. **java.naming.dns.url**
7. **java.naming.authoritative**
8. **java.naming.batchsize**

9. **java.naming.referral**
10. **java.naming.security.protocol**
11. **java.naming.security.authentication**
12. **java.naming.security.principal**
13. **java.naming.security.credentials**
14. **java.naming.language**
15. **java.naming.applet**

For more details on what information to use in these attributes, check your JNDI provider's documentation and consult the Java API reference material.

Example

Example 14.6, "JMS WSDL port specification" shows an example of a JMS WSDL **port** specification.

Example 14.6. JMS WSDL port specification

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

14.2.2. JMS client configuration

Overview

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS **ByteMessage** or a JMS **TextMessage**.

When using an **ByteMessage** the consumer endpoint uses a byte[] as the method for storing data into and retrieving data from the JMS message body. When messages are sent, the message data, including any formating information, is packaged into a byte[] and placed into the message body before it is placed on the wire. When messages are received, the consumer endpoint will attempt to unmarshall the data stored in the message body as if it were packed in a byte[].

When using a **TextMessage**, the consumer endpoint uses a string as the method for storing and retrieving data from the message body. When messages are sent, the message information, including any format-specific information, is converted into a string and placed into the JMS message body.

When messages are received the consumer endpoint will attempt to unmarshall the data stored in the JMS message body as if it were packed into a string.

When native JMS applications interact with Apache CXF consumers, the JMS application is responsible for interpreting the message and the formatting information. For example, if the Apache CXF contract specifies that the binding used for a JMS endpoint is SOAP, and the messages are packaged as **TextMessage**, the receiving JMS application will get a text message containing all of the SOAP envelope information.

Specifying the message type

The type of messages accepted by a JMS consumer endpoint is configured using the optional **jms:client** element. The **jms:client** element is a child of the WSDL **port** element and has one attribute:

Table 14.3. JMS Client WSDL Extensions

messageType	Specifies how the message data will be packaged as a JMS message. **text** specifies that the data will be packaged as a **TextMessage**. **binary** specifies that the data will be packaged as an **ByteMessage**.

Example

Example 14.7, "WSDL for a JMS consumer endpoint" shows the WSDL for configuring a JMS consumer endpoint.

Example 14.7. WSDL for a JMS consumer endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:client messageType="binary" />
  </port>
</service>
```

14.2.3. JMS provider configuration

Overview

JMS provider endpoints have a number of behaviors that are configurable. These include:

- how messages are correlated

- the use of durable subscriptions
- if the service uses local JMS transactions
- the message selectors used by the endpoint

Specifying the configuration

Provider endpoint behaviors are configured using the optional `jms:server` element. The `jms:server` element is a child of the WSDL `wsdl:port` element and has the following attributes:

Table 14.4. JMS provider endpoint WSDL extensions

Attribute	Description
`useMessageIDAsCorrealationID`	Specifies whether JMS will use the message ID to correlate messages. The default is `false`.
`durableSubscriberName`	Specifies the name used to register a durable subscription.
`messageSelector`	Specifies the string value of a message selector to use. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is `false`. [a]

[a] Currently, setting the `transactional` attribute to `true` is not supported by the runtime.

Example

Example 14.8, "WSDL for a JMS provider endpoint" shows the WSDL for configuring a JMS provider endpoint.

Example 14.8. WSDL for a JMS provider endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:server messageSelector="cxf_message_selector"
                useMessageIDAsCorrelationID="true"
```

```
                transactional="true"
                durableSubscriberName="cxf_subscriber" />
  </port>
</service>
```

14.3. USING A NAMED REPLY DESTINATION

Overview

By default, Apache CXF endpoints using JMS create a temporary queue for sending replies back and forth. If you prefer to use named queues, you can configure the queue used to send replies as part of an endpoint's JMS configuration.

Setting the reply destination name

You specify the reply destination using either the **jmsReplyDestinationName** attribute or the **jndiReplyDestinationName** attribute in the endpoint's JMS configuration. A client endpoint will listen for replies on the specified destination and it will specify the value of the attribute in the **ReplyTo** field of all outgoing requests. A service endpoint will use the value of the **jndiReplyDestinationName** attribute as the location for placing replies if there is no destination specified in the request's **ReplyTo** field.

Example

Example 14.9, "JMS Consumer Specification Using a Named Reply Queue" shows the configuration for a JMS client endpoint.

Example 14.9. JMS Consumer Specification Using a Named Reply Queue

```
<jms:conduit name="
{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address destinationStyle="queue"
                 jndiConnectionFactoryName="myConnectionFactory"
                 jndiDestinationName="myDestination"
                 jndiReplyDestinationName="myReplyDestination" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.apache.cxf.transport.jms.MyInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </jms:conduit>
```

APPENDIX A. INTEGRATING WITH APACHE ACTIVEMQ

OVERVIEW

If you are using Apache ActiveMQ as your JMS provider, the JNDI name of your destinations can be specified in a special format that dynamically creates JNDI bindings for queues or topics. This means that it is *not* necessary to configure the JMS provider in advance with the JNDI bindings for your queues or topics.

THE INITIAL CONTEXT FACTORY

The key to integrating Apache ActiveMQ with JNDI is the **`ActiveMQInitialContextFactory`** class. This class is used to create a JNDI **`InitialContext`** instance, which you can then use to access JMS destinations in the JMS broker.

Example A.1, "SOAP/JMS WSDL to connect to Apache ActiveMQ" shows SOAP/JMS WSDL extensions to create a JNDI **`InitialContext`** that is integrated with Apache ActiveMQ.

Example A.1. SOAP/JMS WSDL to connect to Apache ActiveMQ

```
<soapjms:jndiInitialContextFactory>
  org.apache.activemq.jndi.ActiveMQInitialContextFactory
</soapjms:jndiInitialContextFactory>
<soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
```

In Example A.1, "SOAP/JMS WSDL to connect to Apache ActiveMQ" , the Apache ActiveMQ client connects to the broker port located at **`tcp://localhost:61616`**.

LOOKING UP THE CONNECTION FACTORY

As well as creating a JNDI **`InitialContext`** instance, you must specify the JNDI name that is bound to a **`javax.jms.ConnectionFactory`** instance. In the case of Apache ActiveMQ, there is a predefined binding in the **`InitialContext`** instance, which maps the JNDI name **`ConnectionFactory`** to an **`ActiveMQConnectionFactory`** instance. Example A.2, "SOAP/JMS WSDL for specifying the Apache ActiveMQ connection factory" shaows the SOAP/JMS extension element for specifying the Apache ActiveMQ connection factory.

Example A.2. SOAP/JMS WSDL for specifying the Apache ActiveMQ connection factory

```
<soapjms:jndiConnectionFactoryName>
  ConnectionFactory
</soapjms:jndiConnectionFactoryName>
```

SYNTAX FOR DYNAMIC DESTINATIONS

To access queues or topics dynamically, specify the destination's JNDI name as a JNDI composite name in either of the following formats:

```
dynamicQueues/QueueName
```

```
dynamicTopics/TopicName
```

QueueName and *TopicName* are the names that the Apache ActiveMQ broker uses. They are *not* abstract JNDI names.

Example A.3, "WSDL port specification with a dynamically created queue" shows a WSDL port that uses a dynamically created queue.

Example A.3. WSDL port specification with a dynamically created queue

```
<service name="JMSService">
  <port binding="tns:GreeterBinding" name="JMSPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/greeter.request.queue" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

When the application attempts to open the JMS connection, Apache ActiveMQ will check to see if a queue with the JNDI name **`greeter.request.queue`** exists. If it does not exist, it will create a new queue and bind it to the JNDI name **`greeter.request.queue`**.

APPENDIX B. CONDUITS

Abstract

Conduits are a low-level piece of the transport architecture that are used to implement outbound connections. Their behavior and life-cycle can effect system performance and processing load.

OVERVIEW

Conduits manage the client-side, or outbound, transport details in the Apache CXF runtime. They are responsible for opening ports, establishing outbound connections, sending messages, and listening for any responses between an application and a single external endpoint. If an application connects to multiple endpoints, it will have one conduit instance for each endpoint.

Each transport type implements its own conduit using the **`Conduit`** interface. This allows for a standardized interface between the application level functionality and the transports.

In general, you only need to worry about the conduits being used by your application when configuring the client-side transport details. The underlying semantics of how the runtime handles conduits is, generally, not something a developer needs to worry about.

However, there are cases when an understanding of conduit's can prove helpful:

- Implementing a custom transport
- Advanced application tuning to manage limited resources

CONDUIT LIFE-CYCLE

Conduits are managed by the client implementation object. Once created, a conduit lives for the duration of the client implementation object. The conduit's life-cycle is:

1. When the client implementation object is created, it is given a reference to a **`ConduitSelector`** object.
2. When the client needs to send a message is request's a reference to a conduit from the conduit selector.

   If the message is for a new endpoint, the conduit selector creates a new conduit and passes it to the client implementation. Otherwise, it passes the client a reference to the conduit for the target endpoint.
3. The conduit sends messages when needed.
4. When the client implementation object is destroyed, all of the conduits associated with it are destroyed.

CONDUIT WEIGHT

The weight of a conduit object depends on the transport implementation. HTTP conduits are extremely light weight. JMS conduits are heavy because they are associated with the JMS **`Session`** object and one or more **`JMSListenerContainer`** objects.

PART IV. CONFIGURING WEB SERVICE ENDPOINTS

Abstract

This guide describes how to create Apache CXF endpoints in Red Hat JBoss Fuse.

CHAPTER 15. CONFIGURING JAX-WS ENDPOINTS

Abstract

JAX-WS endpoints are configured using one of three Spring configuration elements. The correct element depends on what type of endpoint you are configuring and which features you wish to use. For consumers you use the `jaxws:client` element. For service providers you can use either the `jaxws:endpoint` element or the `jaxws:server` element.

The information used to define an endpoint is typically defined in the endpoint's contract. You can use the configuration element's to override the information in the contract. You can also use the configuration elements to provide information that is not provided in the contract.

NOTE

When dealing with endpoints developed using a Java-first approach it is likely that the SEI serving as the endpoint's contract is lacking information about the type of binding and transport to use.

You must use the configuration elements to activate advanced features such as WS-RM. This is done by providing child elements to the endpoint's configuration element.

15.1. CONFIGURING SERVICE PROVIDERS

Apache CXF has two elements that can be used to configure a service provider:

- Section 15.1.1, “Using the jaxws:endpoint Element”
- Section 15.1.2, “Using the jaxws:server Element”

The differences between the two elements are largely internal to the runtime. The `jaxws:endpoint` element injects properties into the `org.apache.cxf.jaxws.EndpointImpl` object created to support a service endpoint. The `jaxws:server` element injects properties into the `org.apache.cxf.jaxws.support.JaxWsServerFactoryBean` object created to support the endpoint. The `EndpointImpl` object passes the configuration data to the `JaxWsServerFactoryBean` object. The `JaxWsServerFactoryBean` object is used to create the actual service object. Because either configuration element will configure a service endpoint, you can choose based on the syntax you prefer.

15.1.1. Using the jaxws:endpoint Element

Overview

The `jaxws:endpoint` element is the default element for configuring JAX-WS service providers. Its attributes and children specify all of the information needed to instantiate a service provider. Many of the attributes map to information in the service's contract. The children are used to configure interceptors and other advanced features.

Identifying the endpoint being configured

For the runtime to apply the configuration to the proper service provider, it must be able to identify it. The basic means for identifying a service provider is to specify the class that implements the endpoint. This is done using the **jaxws:endpoint** element's **implementor** attribute.

For instances where different endpoint's share a common implementation, it is possible to provide different configuration for each endpoint. There are two approaches for distinguishing a specific endpoint in configuration:

- a combination of the **serviceName** attribute and the **endpointName** attribute

  The **serviceName** attribute specifies the **wsdl:service** element defining the service's endpoint. The **endpointName** attribute specifies the specific **wsdl:port** element defining the service's endpoint. Both attributes are specified as QNames using the format ***ns*:*name***. *ns* is the namespace of the element and *name* is the value of the element's **name** attribute.

  TIP

  If the **wsdl:service** element only has one **wsdl:port** element, the **endpointName** attribute can be omitted.

- the **name** attribute

  The **name** attribute specifies the QName of the specific **wsdl:port** element defining the service's endpoint. The QName is provided in the format ***{ns}localPart***. *ns* is the namespace of the **wsdl:port** element and *localPart* is the value of the **wsdl:port** element's **name** attribute.

Attributes

The attributes of the **jaxws:endpoint** element configure the basic properties of the endpoint. These properties include the address of the endpoint, the class that implements the endpoint, and the **bus** that hosts the endpoint.

Table 15.1, "Attributes for Configuring a JAX-WS Service Provider Using the **jaxws:endpoint** Element" describes the attribute of the **jaxws:endpoint** element.

Table 15.1. Attributes for Configuring a JAX-WS Service Provider Using the jaxws:endpoint Element

Attribute	Description
id	Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
implementor	Specifies the class implementing the service. You can specify the implementation class using either the class name or an ID reference to a Spring bean configuring the implementation class. This class must be on the classpath.

Attribute	Description
implementorClass	Specifies the class implementing the service. This attribute is useful when the value provided to the **implementor** attribute is a reference to a bean that is wrapped using Spring AOP.
address	Specifies the address of an HTTP endpoint. This value overrides the value specified in the services contract.
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the service is deployed.
endpointName	Specifies the value of the service's **wsdl:port** element's **name** attribute. It is specified as a QName using the format ***ns:name*** where *ns* is the namespace of the **wsdl:port** element.
serviceName	Specifies the value of the service's **wsdl:service** element's **name** attribute. It is specified as a QName using the format ***ns:name*** where *ns* is the namespace of the **wsdl:service** element.
publish	Specifies if the service should be automatically published. If this is set to **false**, the developer must explicitly publish the endpoint.
bus	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful when configuring several endpoints to use a common set of features.
bindingUri	Specifies the ID of the message binding the service uses. A list of valid binding IDs is provided in Appendix C, *Apache CXF Binding IDs*.
name	Specifies the stringified QName of the service's **wsdl:port** element. It is specified as a QName using the format ***{ns}localPart***. *ns* is the namespace of the **wsdl:port** element and *localPart* is the value of the **wsdl:port** element's **name** attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is **false**. Setting this to **true** instructs the bean factory not to instantiate the bean.

Attribute	Description
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before it can be instantiated.
createdFromAPI	Specifies that the user created that bean using Apache CXF APIs, such as **Endpoint.publish()** or **Service.getPort()**. The default is **false**. Setting this to **true** does the following: • Changes the internal name of the bean by appending **.jaxws-endpoint** to its id • Makes the bean abstract
publishedEndpointUrl	The URL that is placed in the **address** element of the generated WSDL. If this value is not specified, the value of the **address** attribute is used. This attribute is useful when the "public" URL is not be the same as the URL on which the service is deployed.

In addition to the attributes listed in Table 15.1, "Attributes for Configuring a JAX-WS Service Provider Using the **jaxws:endpoint** Element", you might need to use multiple **xmlns:*shortName*** attributes to declare the namespaces used by the **endpointName** and **serviceName** attributes.

Example

Example 15.1, "Simple JAX-WS Endpoint Configuration" shows the configuration for a JAX-WS endpoint that specifies the address where the endpoint is published. The example assumes that you want to use the defaults for all other values or that the implementation has specified values in the annotations.

Example 15.1. Simple JAX-WS Endpoint Configuration

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="example"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo" />
</beans>
```

Example 15.2, "JAX-WS Endpoint Configuration with a Service Name" shows the configuration for a JAX-WS endpoint whose contract contains two service definitions. In this case, you must specify which service definition to instantiate using the **serviceName** attribute.

Example 15.2. JAX-WS Endpoint Configuration with a Service Name

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">

  <jaxws:endpoint id="example2"
                  implementor="org.apache.cxf.example.DemoImpl"
                  serviceName="samp:demoService2"
                  xmlns:samp="http://org.apache.cxf/wsdl/example" />

</beans>
```

The **xmlns:samp** attribute specifies the namespace in which the WSDL **service** element is defined.

15.1.2. Using the jaxws:server Element

Overview

The **jaxws:server** element is an element for configuring JAX-WS service providers. It injects the configuration information into the **org.apache.cxf.jaxws.support.JaxWsServerFactoryBean**. This is a Apache CXF specific object. If you are using a pure Spring approach to building your services, you will not be forced to use Apache CXF specific APIs to interact with the service.

The attributes and children of the **jaxws:server** element specify all of the information needed to instantiate a service provider. The attributes specify the information that is required to instantiate an endpoint. The children are used to configure interceptors and other advanced features.

Identifying the endpoint being configured

In order for the runtime to apply the configuration to the proper service provider, it must be able to identify it. The basic means for identifying a service provider is to specify the class that implements the endpoint. This is done using the **jaxws:server** element's **serviceBean** attribute.

For instances where different endpoint's share a common implementation, it is possible to provide different configuration for each endpoint. There are two approaches for distinguishing a specific endpoint in configuration:

- a combination of the **serviceName** attribute and the **endpointName** attribute

  The **serviceName** attribute specifies the **wsdl:service** element defining the service's endpoint. The **endpointName** attribute specifies the specific **wsdl:port** element defining the service's endpoint. Both attributes are specified as QNames using the format ***ns*:*name***. *ns* is the namespace of the element and *name* is the value of the element's **name** attribute.

TIP

If the **wsdl:service** element only has one **wsdl:port** element, the **endpointName** attribute can be omitted.

- the **name** attribute

  The **name** attribute specifies the QName of the specific **wsdl:port** element defining the service's endpoint. The QName is provided in the format ***{ns}localPart***. *ns* is the namespace of the **wsdl:port** element and *localPart* is the value of the **wsdl:port** element's **name** attribute.

Attributes

The attributes of the **jaxws:server** element configure the basic properties of the endpoint. These properties include the address of the endpoint, the class that implements the endpoint, and the **bus** that hosts the endpoint.

Table 15.2, "Attributes for Configuring a JAX-WS Service Provider Using the **jaxws:server** Element" describes the attribute of the **jaxws:server** element.

Table 15.2. Attributes for Configuring a JAX-WS Service Provider Using the jaxws:server Element

Attribute	Description
id	Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
serviceBean	Specifies the class implementing the service. You can specify the implementation class using either the class name or an ID reference to a Spring bean configuring the implementation class. This class must be on the classpath.
serviceClass	Specifies the class implementing the service. This attribute is useful when the value provided to the **implementor** attribute is a reference to a bean that is wrapped using Spring AOP.
address	Specifies the address of an HTTP endpoint. This value will override the value specified in the services contract.
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the service is deployed.
endpointName	Specifies the value of the service's **wsdl:port** element's **name** attribute. It is specified as a QName using the format ***ns:name***, where *ns* is the namespace of the **wsdl:port** element.

Attribute	Description
serviceName	Specifies the value of the service's **wsdl:service** element's **name** attribute. It is specified as a QName using the format ***ns:name***, where *ns* is the namespace of the **wsdl:service** element.
start	Specifies if the service should be automatically published. If this is set to **false**, the developer must explicitly publish the endpoint.
bus	Specifies the ID of the Spring bean configuring the bus used to manage the service endpoint. This is useful when configuring several endpoints to use a common set of features.
bindingId	Specifies the ID of the message binding the service uses. A list of valid binding IDs is provided in Appendix C, *Apache CXF Binding IDs*.
name	Specifies the stringified QName of the service's **wsdl:port** element. It is specified as a QName using the format ***{ns}localPart***, where *ns* is the namespace of the **wsdl:port** element and *localPart* is the value of the **wsdl:port** element's **name** attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is **false**. Setting this to **true** instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before the endpoint can be instantiated.
createdFromAPI	Specifies that the user created that bean using Apache CXF APIs, such as **Endpoint.publish()** or **Service.getPort()**. The default is **false**. Setting this to **true** does the following: • Changes the internal name of the bean by appending **.jaxws-endpoint** to its id • Makes the bean abstract

In addition to the attributes listed in Table 15.2, "Attributes for Configuring a JAX-WS Service Provider Using the `jaxws:server` Element", you might need to use multiple `xmlns:shortName` attributes to declare the namespaces used by the `endpointName` and `serviceName` attributes.

Example

Example 15.3, "Simple JAX-WS Server Configuration" shows the configuration for a JAX-WS endpoint that specifies the address where the endpoint is published.

Example 15.3. Simple JAX-WS Server Configuration

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:server id="exampleServer"
                serviceBean="org.apache.cxf.example.DemoImpl"
                address="http://localhost:8080/demo" />
</beans>
```

15.1.3. Adding Functionality to Service Providers

Overview

The `jaxws:endpoint` and the `jaxws:server` elements provide the basic configuration information needed to instantiate a service provider. To add functionality to your service provider or to perform advanced configuration you must add child elements to the configuration.

Child elements allow you to do the following:

- Change the endpoint's logging behavior
- Add interceptors to the endpoint's messaging chain
- Enable WS-Addressing features
- Enable reliable messaging

Elements

Table 15.3, "Elements Used to Configure JAX-WS Service Providers" describes the child elements that `jaxws:endpoint` supports.

Table 15.3. Elements Used to Configure JAX-WS Service Providers

Element	Description
`jaxws:handlers`	Specifies a list of JAX-WS `Handler` implementations for processing messages.

Element	Description
`jaxws:inInterceptors`	Specifies a list of interceptors that process inbound requests. For more information see Part VII, "Developing Apache CXF Interceptors".
`jaxws:inFaultInterceptors`	Specifies a list of interceptors that process inbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors".
`jaxws:outInterceptors`	Specifies a list of interceptors that process outbound replies. For more information see Part VII, "Developing Apache CXF Interceptors".
`jaxws:outFaultInterceptors`	Specifies a list of interceptors that process outbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors".
`jaxws:binding`	Specifies a bean configuring the message binding used by the endpoint. Message bindings are configured using implementations of the `org.apache.cxf.binding.BindingFactory` interface.[a]
`jaxws:dataBinding` [b]	Specifies the class implementing the data binding used by the endpoint. This is specified using an embedded bean definition.
`jaxws:executor`	Specifies a Java executor that is used for the service. This is specified using an embedded bean definition.
`jaxws:features`	Specifies a list of beans that configure advanced features of Apache CXF. You can provide either a list of bean references or a list of embedded beans.
`jaxws:invoker`	Specifies an implementation of the `org.apache.cxf.service.Invoker` interface used by the service. [c]
`jaxws:properties`	Specifies a Spring map of properties that are passed along to the endpoint. These properties can be used to control features like enabling MTOM support.
`jaxws:serviceFactory`	Specifies a bean configuring the `JaxWsServiceFactoryBean` object used to instantiate the service.

Element	Description

[a] The SOAP binding is configured using the `soap:soapBinding` bean.

[b] The `jaxws:endpoint` element does not support the `jaxws:dataBinding` element.

[c] The `Invoker` implementation controls how a service is invoked. For example, it controls whether each request is handled by a new instance of the service implementation or if state is preserved across invocations.

15.2. CONFIGURING CONSUMER ENDPOINTS

Overview

JAX-WS consumer endpoints are configured using the `jaxws:client` element. The element's attributes provide the basic information necessary to create a consumer.

To add other functionality, like WS-RM, to the consumer you add children to the `jaxws:client` element. Child elements are also used to configure the endpoint's logging behavior and to inject other properties into the endpoint's implementation.

Basic Configuration Properties

The attributes described in Table 15.4, "Attributes Used to Configure a JAX-WS Consumer" provide the basic information necessary to configure a JAX-WS consumer. You only need to provide values for the specific properties you want to configure. Most of the properties have sensible defaults, or they rely on information provided by the endpoint's contract.

Table 15.4. Attributes Used to Configure a JAX-WS Consumer

Attribute	Description
`address`	Specifies the HTTP address of the endpoint where the consumer will make requests. This value overrides the value set in the contract.
`bindingId`	Specifies the ID of the message binding the consumer uses. A list of valid binding IDs is provided in Appendix C, *Apache CXF Binding IDs*.
`bus`	Specifies the ID of the Spring bean configuring the bus managing the endpoint.
`endpointName`	Specifies the value of the`wsdl:port` element's `name` attribute for the service on which the consumer is making requests. It is specified as a QName using the format ***`ns:name`***, where *ns* is the namespace of the `wsdl:port` element.

Attribute	Description
serviceName	Specifies the value of the **wsdl:service** element's **name** attribute for the service on which the consumer is making requests. It is specified as a QName using the format ***ns:name*** where *ns* is the namespace of the **wsdl:service** element.
username	Specifies the username used for simple username/password authentication.
password	Specifies the password used for simple username/password authentication.
serviceClass	Specifies the name of the service endpoint interface(SEI).
wsdlLocation	Specifies the location of the endpoint's WSDL contract. The WSDL contract's location is relative to the folder from which the client is deployed.
name	Specifies the stringified QName of the **wsdl:port** element for the service on which the consumer is making requests. It is specified as a QName using the format ***{ns}localPart***, where *ns* is the namespace of the **wsdl:port** element and *localPart* is the value of the **wsdl:port** element's **name** attribute.
abstract	Specifies if the bean is an abstract bean. Abstract beans act as parents for concrete bean definitions and are not instantiated. The default is **false**. Setting this to **true** instructs the bean factory not to instantiate the bean.
depends-on	Specifies a list of beans that the endpoint depends on being instantiated before it can be instantiated.
createdFromAPI	Specifies that the user created that bean using Apache CXF APIs like **Service.getPort()**. The default is **false**. Setting this to **true** does the following: • Changes the internal name of the bean by appending **.jaxws-client** to its id • Makes the bean abstract

In addition to the attributes listed in Table 15.4, "Attributes Used to Configure a JAX-WS Consumer" , it might be necessary to use multiple `xmlns:shortName` attributes to declare the namespaces used by the `endpointName` and the `serviceName` attributes.

Adding functionality

To add functionality to your consumer or to perform advanced configuration, you must add child elements to the configuration.

Child elements allow you to do the following:

- Change the endpoint's logging behavior
- Add interceptors to the endpoint's messaging chain
- Enable WS-Addressing features
- Enable reliable messaging

Table 15.5, "Elements For Configuring a Consumer Endpoint" describes the child element's you can use to configure a JAX-WS consumer.

Table 15.5. Elements For Configuring a Consumer Endpoint

Element	Description
`jaxws:binding`	Specifies a bean configuring the message binding used by the endpoint. Message bindings are configured using implementations of the `org.apache.cxf.binding.BindingFactory` interface.[a]
`jaxws:dataBinding`	Specifies the class implementing the data binding used by the endpoint. You specify this using an embedded bean definition. The class implementing the JAXB data binding is `org.apache.cxf.jaxb.JAXBDataBinding`.
`jaxws:features`	Specifies a list of beans that configure advanced features of Apache CXF. You can provide either a list of bean references or a list of embedded beans.
`jaxws:handlers`	Specifies a list of JAX-WS `Handler` implementations for processing messages.
`jaxws:inInterceptors`	Specifies a list of interceptors that process inbound responses. For more information see Part VII, "Developing Apache CXF Interceptors".
`jaxws:inFaultInterceptors`	Specifies a list of interceptors that process inbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors".

Element	Description
jaxws:outInterceptors	Specifies a list of interceptors that process outbound requests. For more information see Part VII, "Developing Apache CXF Interceptors".
jaxws:outFaultInterceptors	Specifies a list of interceptors that process outbound fault messages. For more information see Part VII, "Developing Apache CXF Interceptors".
jaxws:properties	Specifies a map of properties that are passed to the endpoint.
jaxws:conduitSelector	Specifies an **org.apache.cxf.endpoint.ConduitSelector** implementation for the client to use. A **ConduitSelector** implementation will override the default process used to select the **Conduit** object that is used to process outbound requests.

[a] The SOAP binding is configured using the **soap:soapBinding** bean.

Example

Example 15.4, "Simple Consumer Configuration" shows a simple consumer configuration.

Example 15.4. Simple Consumer Configuration

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:client id="bookClient"
                serviceClass="org.apache.cxf.demo.BookClientImpl"
                address="http://localhost:8080/books"/>
  ...
</beans>
```

CHAPTER 16. APACHE CXF LOGGING

Abstract

This chapter describes how to configure logging in the Apache CXF runtime.

16.1. OVERVIEW OF APACHE CXF LOGGING

Overview

Apache CXF uses the Java logging utility, **`java.util.logging`**. Logging is configured in a logging configuration file that is written using the standard **`java.util.Properties`** format. To run logging on an application, you can specify logging programmatically or by defining a property at the command that points to the logging configuration file when you start the application.

Default `logging.properties` file

Apache CXF comes with a default **`logging.properties`** file, which is located in your *InstallDir*`/etc` directory. This file configures both the output destination for the log messages and the message level that is published. The default configuration sets the loggers to print message flagged with the **`WARNING`** level to the console. You can either use the default file without changing any of the configuration settings or you can change the configuration settings to suit your specific application.

Logging feature

Apache CXF includes a logging feature that can be plugged into your client or your service to enable logging. Example 16.1, “Configuration for Enabling Logging” shows the configuration to enable the logging feature.

Example 16.1. Configuration for Enabling Logging

```
<jaxws:endpoint...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

For more information, see Section 16.6, “Logging Message Content”.

Where to begin?

To run a simple example of logging follow the instructions outlined in a Section 16.2, “Simple Example of Using Logging”.

For more information on how logging works in Apache CXF, read this entire chapter.

More information on `java.util.logging`

The **java.util.logging** utility is one of the most widely used Java logging frameworks. There is a lot of information available online that describes how to use and extend this framework. As a starting point, however, the following documents gives a good overview of **java.util.logging**:

- http://java.sun.com/j2se/1.5.0/docs/guide/logging/overview.html
- http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/package-summary.html

16.2. SIMPLE EXAMPLE OF USING LOGGING

Changing the log levels and output destination

To change the log level and output destination of the log messages in the wsdl_first sample application, complete the following steps:

1. Run the sample server as described in the *Running the demo using java*section of the **README.txt** file in the ***InstallDir*/samples/wsdl_first** directory. Note that the **server start** command specifies the default **logging.properties** file, as follows:

Platform	Command
Windows	**start java -Djava.util.logging.config.file=%CXF_HOME%\etc\logging.properties demo.hw.server.Server**
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/logging.properties demo.hw.server.Server &**

   The default **logging.properties** file is located in the ***InstallDir*/etc** directory. It configures the Apache CXF loggers to print **WARNING** level log messages to the console. As a result, you see very little printed to the console.

2. Stop the server as described in the **README.txt** file.

3. Make a copy of the default **logging.properties** file, name it **mylogging.properties** file, and save it in the same directory as the default **logging.properties** file.

4. Change the global logging level and the console logging levels in your **mylogging.properties** file to **INFO** by editing the following lines of configuration:

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

5. Restart the server using the following command:

Platform	Command
Windows	`start java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server`
UNIX	`java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &`

Because you configured the global logging and the console logger to log messages of level **INFO**, you see a lot more log messages printed to the console.

16.3. DEFAULT LOGGING CONFIGURATION FILE

The default logging configuration file, **logging.properties**, is located in the ***InstallDir*/etc** directory. It configures the Apache CXF loggers to print **WARNING** level messages to the console. If this level of logging is suitable for your application, you do not have to make any changes to the file before using it. You can, however, change the level of detail in the log messages. For example, you can change whether log messages are sent to the console, to a file or to both. In addition, you can specify logging at the level of individual packages.

NOTE

This section discusses the configuration properties that appear in the default **logging.properties** file. There are, however, many other **java.util.logging** configuration properties that you can set. For more information on the **java.util.logging** API, see the **java.util.logging** javadoc at: http://java.sun.com/j2se/1.5/docs/api/java/util/logging/package-summary.html.

16.3.1. Configuring Logging Output

The Java logging utility, **java.util.logging**, uses handler classes to output log messages. Table 16.1, "Java.util.logging Handler Classes" shows the handlers that are configured in the default **logging.properties** file.

Table 16.1. Java.util.logging Handler Classes

Handler Class	Outputs to
ConsoleHandler	Outputs log messages to the console
FileHandler	Outputs log messages to a file

IMPORTANT

The handler classes must be on the system classpath in order to be installed by the Java VM when it starts. This is done when you set the Apache CXF environment.

Configuring the console handler

Example 16.2, "Configuring the Console Handler" shows the code for configuring the console logger.

Example 16.2. Configuring the Console Handler

```
handlers= java.util.logging.ConsoleHandler
```

The console handler also supports the configuration properties shown in Example 16.3, "Console Handler Properties".

Example 16.3. Console Handler Properties

```
1 java.util.logging.ConsoleHandler.level = WARNING
  java.util.logging.ConsoleHandler.formatter =
2 java.util.logging.SimpleFormatter
```

The configuration properties shown in Example 16.3, "Console Handler Properties" can be explained as follows:

1. The console handler supports a separate log level configuration property. This allows you to limit the log messages printed to the console while the global logging setting can be different (see Section 16.3.2, "Configuring Logging Levels"). The default setting is **`WARNING`**.

2. Specifies the **`java.util.logging`** formatter class that the console handler class uses to format the log messages. The default setting is the **`java.util.logging.SimpleFormatter`**.

Configuring the file handler

Example 16.4, "Configuring the File Handler" shows code that configures the file handler.

Example 16.4. Configuring the File Handler

```
handlers= java.util.logging.FileHandler
```

The file handler also supports the configuration properties shown in Example 16.5, "File Handler Configuration Properties".

Example 16.5. File Handler Configuration Properties

```
1 java.util.logging.FileHandler.pattern = %h/java%u.log
2 java.util.logging.FileHandler.limit = 50000
3 java.util.logging.FileHandler.count = 1
  java.util.logging.FileHandler.formatter =
4 java.util.logging.XMLFormatter
```

The configuration properties shown in Example 16.5, "File Handler Configuration Properties" can be explained as follows:

1. Specifies the location and pattern of the output file. The default setting is your home directory.
2. Specifies, in bytes, the maximum amount that the logger writes to any one file. The default setting is **50000**. If you set it to zero, there is no limit on the amount that the logger writes to any one file.
3. Specifies how many output files to cycle through. The default setting is **1**.
4. Specifies the **java.util.logging** formatter class that the file handler class uses to format the log messages. The default setting is the **java.util.logging.XMLFormatter**.

Configuring both the console handler and the file handler

You can set the logging utility to output log messages to both the console and to a file by specifying the console handler and the file handler, separated by a comma, as shown in Example 16.6, "Configuring Both Console Logging and File Logging".

Example 16.6. Configuring Both Console Logging and File Logging

```
handlers= java.util.logging.FileHandler,
java.util.logging.ConsoleHandler
```

16.3.2. Configuring Logging Levels

Logging levels

The **java.util.logging** framework supports the following levels of logging, from the least verbose to the most verbose:

- **SEVERE**
- **WARNING**
- **INFO**
- **CONFIG**
- **FINE**
- **FINER**
- **FINEST**

Configuring the global logging level

To configure the types of event that are logged across all loggers, configure the global logging level as shown in Example 16.7, "Configuring Global Logging Levels".

Example 16.7. Configuring Global Logging Levels

```
.level= WARNING
```

Configuring logging at an individual package level

The **java.util.logging** framework supports configuring logging at the level of an individual package. For example, the line of code shown in Example 16.8, "Configuring Logging at the Package Level" configures logging at a **SEVERE** level on classes in the com.xyz.foo package.

Example 16.8. Configuring Logging at the Package Level

```
com.xyz.foo.level = SEVERE
```

16.4. ENABLING LOGGING AT THE COMMAND LINE

Overview

You can run the logging utility on an application by defining a **java.util.logging.config.file** property when you start the application. You can either specify the default **logging.properties** file or a **logging.properties** file that is unique to that application.

Specifying the log configuration file on application start-up

To specify logging on application start-up add the flag shown in Example 16.9, "Flag to Start Logging on the Command Line" when starting the application.

Example 16.9. Flag to Start Logging on the Command Line

```
-Djava.util.logging.config.file=myfile
```

16.5. LOGGING FOR SUBSYSTEMS AND SERVICES

You can use the **com.xyz.foo.level** configuration property described in the section called "Configuring logging at an individual package level" to set fine-grained logging for specified Apache CXF logging subsystems.

Apache CXF logging subsystems

Table 16.2, "Apache CXF Logging Subsystems" shows a list of available Apache CXF logging subsystems.

Table 16.2. Apache CXF Logging Subsystems

Subsystem	Description
org.apache.cxf.aegis	Aegis binding

Subsystem	Description
org.apache.cxf.binding.coloc	colocated binding
org.apache.cxf.binding.http	HTTP binding
org.apache.cxf.binding.jbi	JBI binding
org.apache.cxf.binding.object	Java Object binding
org.apache.cxf.binding.soap	SOAP binding
org.apache.cxf.binding.xml	XML binding
org.apache.cxf.bus	Apache CXF bus
org.apache.cxf.configuration	configuration framework
org.apache.cxf.endpoint	server and client endpoints
org.apache.cxf.interceptor	interceptors
org.apache.cxf.jaxws	Front-end for JAX-WS style message exchange, JAX-WS handler processing, and interceptors relating to JAX-WS and configuration
org.apache.cxf.jbi	JBI container integration classes
org.apache.cxf.jca	JCA container integration classes
org.apache.cxf.js	JavaScript front-end
org.apache.cxf.transport.http	HTTP transport
org.apache.cxf.transport.https	secure version of HTTP transport, using HTTPS
org.apache.cxf.transport.jbi	JBI transport
org.apache.cxf.transport.jms	JMS transport
org.apache.cxf.transport.local	transport implementation using local file system
org.apache.cxf.transport.servlet	HTTP transport and servlet implementation for loading JAX-WS endpoints into a servlet container
org.apache.cxf.ws.addressing	WS-Addressing implementation

Subsystem	Description
`org.apache.cxf.ws.policy`	WS-Policy implementation
`org.apache.cxf.ws.rm`	WS-ReliableMessaging (WS-RM) implementation
`org.apache.cxf.ws.security.wss4j`	WSS4J security implementation

Example

The WS-Addressing sample is contained in the ***`InstallDir`*`/samples/ws_addressing`** directory. Logging is configured in the **`logging.properties`** file located in that directory. The relevant lines of configuration are shown in Example 16.10, "Configuring Logging for WS-Addressing".

Example 16.10. Configuring Logging for WS-Addressing

```
java.util.logging.ConsoleHandler.formatter =
demos.ws_addressing.common.ConciseFormatter
...
org.apache.cxf.ws.addressing.soap.MAPCodec.level = INFO
```

The configuration in Example 16.10, "Configuring Logging for WS-Addressing" enables the snooping of log messages relating to WS-Addressing headers, and displays them to the console in a concise form.

For information on running this sample, see the **`README.txt`** file located in the ***`InstallDir`*`/samples/ws_addressing`** directory.

16.6. LOGGING MESSAGE CONTENT

You can log the content of the messages that are sent between a service and a consumer. For example, you might want to log the contents of SOAP messages that are being sent between a service and a consumer.

Configuring message content logging

To log the messages that are sent between a service and a consumer, and vice versa, complete the following steps:

1. Add the logging feature to your endpoint's configuration.
2. Add the logging feature to your consumer's configuration.
3. Configure the logging system log **`INFO`** level messages.

Adding the logging feature to an endpoint

Add the logging feature your endpoint's configuration as shown in Example 16.11, "Adding Logging to Endpoint Configuration".

Example 16.11. Adding Logging to Endpoint Configuration

```
<jaxws:endpoint ...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

The example XML shown in Example 16.11, “Adding Logging to Endpoint Configuration” enables the logging of SOAP messages.

Adding the logging feature to a consumer

Add the logging feature your client's configuration as shown in Example 16.12, “Adding Logging to Client Configuration”.

Example 16.12. Adding Logging to Client Configuration

```
<jaxws:client ...>
   <jaxws:features>
      <bean class="org.apache.cxf.feature.LoggingFeature"/>
    </jaxws:features>
</jaxws:client>
```

The example XML shown in Example 16.12, “Adding Logging to Client Configuration” enables the logging of SOAP messages.

Set logging to log INFO level messages

Ensure that the **logging.properties** file associated with your service is configured to log **INFO** level messages, as shown in Example 16.13, “Setting the Logging Level to INFO”.

Example 16.13. Setting the Logging Level to INFO

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

Logging SOAP messages

To see the logging of SOAP messages modify the wsdl_first sample application located in the ***InstallDir*/samples/wsdl_first** directory, as follows:

1. Add the **jaxws:features** element shown in Example 16.14, “Endpoint Configuration for Logging SOAP Messages” to the **cxf.xml** configuration file located in the wsdl_first sample's directory:

   Example 16.14. Endpoint Configuration for Logging SOAP Messages

   ```
   <jaxws:endpoint name="
   {http://apache.org/hello_world_soap_http}SoapPort"
                   createdFromAPI="true">
   ```

```
    <jaxws:properties>
      <entry key="schema-validation-enabled" value="true" />
    </jaxws:properties>
    <jaxws:features>
      <bean class="org.apache.cxf.feature.LoggingFeature"/>
    </jaxws:features>
  </jaxws:endpoint>
```

2. The sample uses the default **logging.properties** file, which is located in the ***InstallDir*/etc** directory. Make a copy of this file and name it **mylogging.properties**.

3. In the **mylogging.properties** file, change the logging levels to **INFO** by editing the **.level** and the **java.util.logging.ConsoleHandler.level** configuration properties as follows:

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

4. Start the server using the new configuration settings in both the **cxf.xml** file and the **mylogging.properties** file as follows:

Platform	Command
Windows	**start java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server**
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &**

5. Start the hello world client using the following command:

Platform	Command
Windows	**java -Djava.util.logging.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.client.Client .\wsdl\hello_world.wsdl**
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.client.Client ./wsdl/hello_world.wsdl**

The SOAP messages are logged to the console.

CHAPTER 17. DEPLOYING WS-ADDRESSING

Abstract

Apache CXF supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the Apache CXF runtime environment.

17.1. INTRODUCTION TO WS-ADDRESSING

Overview

WS-Addressing is a specification that allows services to communicate addressing information in a transport neutral way. It consists of two parts:

- A structure for communicating a reference to a Web service endpoint
- A set of Message Addressing Properties (MAP) that associate addressing information with a particular message

Supported specifications

Apache CXF supports both the WS-Addressing 2004/08 specification and the WS-Addressing 2005/03 specification.

Further information

For detailed information on WS-Addressing, see the 2004/08 submission at http://www.w3.org/Submission/ws-addressing/.

17.2. WS-ADDRESSING INTERCEPTORS

Overview

In Apache CXF, WS-Addressing functionality is implemented as interceptors. The Apache CXF runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the WS-Addressing interceptors are added to the application's interceptor chain, any WS-Addressing information included with a message is processed.

WS-Addressing Interceptors

The WS-Addressing implementation consists of two interceptors, as described in Table 17.1, “WS-Addressing Interceptors”.

Table 17.1. WS-Addressing Interceptors

Interceptor	Description

Interceptor	Description
`org.apache.cxf.ws.addressing.MAPAggregator`	A logical interceptor responsible for aggregating the Message Addressing Properties (MAPs) for outgoing messages.
`org.apache.cxf.ws.addressing.soap.MAPCodec`	A protocol-specific interceptor responsible for encoding and decoding the Message Addressing Properties (MAPs) as SOAP headers.

17.3. ENABLING WS-ADDRESSING

Overview

To enable WS-Addressing the WS-Addressing interceptors must be added to the inbound and outbound interceptor chains. This is done in one of the following ways:

- Apache CXF Features
- RMAssertion and WS-Policy Framework
- Using Policy Assertion in a WS-Addressing Feature

Adding WS-Addressing as a Feature

WS-Addressing can be enabled by adding the WS-Addressing feature to the client and the server configuration as shown in Example 17.1, “client.xml—Adding WS-Addressing Feature to Client Configuration” and Example 17.2, “server.xml—Adding WS-Addressing Feature to Server Configuration” respectively.

Example 17.1. client.xml—Adding WS-Addressing Feature to Client Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:client ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:client>
</beans>
```

Example 17.2. server.xml—Adding WS-Addressing Feature to Server Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

17.4. CONFIGURING WS-ADDRESSING ATTRIBUTES

Overview

The Apache CXF WS-Addressing feature element is defined in the namespace **`http://cxf.apache.org/ws/addressing`**. It supports the two attributes described in Table 17.2, "WS-Addressing Attributes".

Table 17.2. WS-Addressing Attributes

Attribute Name	Value
`allowDuplicates`	A boolean that determines if duplicate MessageIDs are tolerated. The default setting is **`true`**.
`usingAddressingAdvisory`	A boolean that indicates if the presence of the **`UsingAddressing`** element in the WSDL is advisory only; that is, its absence does not prevent the encoding of WS-Addressing headers.

Configuring WS-Addressing attributes

Configure WS-Addressing attributes by adding the attribute and the value you want to set it to the WS-Addressing feature in your server or client configuration file. For example, the following configuration extract sets the **`allowDublicates`** attribute to **`false`** on the server endpoint:

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing" ...>
    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing allowDuplicates="false"/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Using a WS-Policy assertion embedded in a feature

In Example 17.3, "Using the Policies to Configure WS-Addressing" an addressing policy assertion to enable non-anonymous responses is embedded in the **policies** element.

Example 17.3. Using the Policies to Configure WS-Addressing

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:policy="http://cxf.apache.org/policy-config"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-
policy.xsd
http://cxf.apache.org/ws/addressing
http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint name="
{http://cxf.apache.org/greeter_control}GreeterPort"
                    createdFromAPI="true">
        <jaxws:features>
            <policy:policies>
                <wsp:Policy
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                    <wsam:Addressing>
                        <wsp:Policy>
                            <wsam:NonAnonymousResponses/>
                        </wsp:Policy>
                    </wsam:Addressing>
                </wsp:Policy>
            <policy:policies>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

CHAPTER 18. ENABLING RELIABLE MESSAGING

Abstract

Apache CXF supports WS-Reliable Messaging(WS-RM). This chapter explains how to enable and configure WS-RM in Apache CXF.

18.1. INTRODUCTION TO WS-RM

Overview

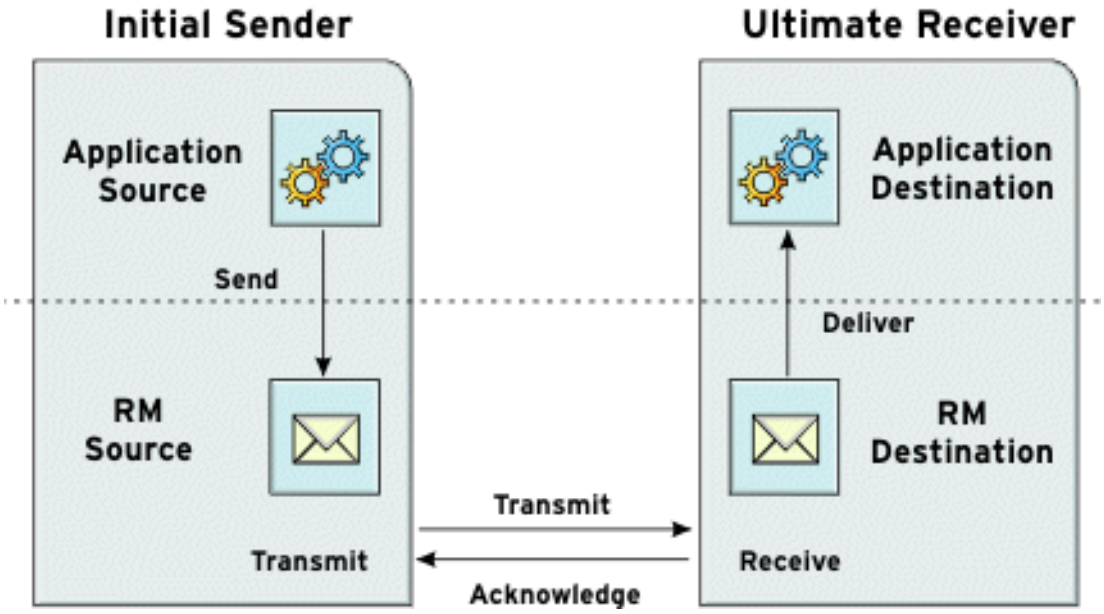
WS-ReliableMessaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

For example, WS-RM can be used to ensure that the correct messages have been delivered across a network exactly once, and in the correct order.

How WS-RM works

WS-RM ensures the reliable delivery of messages between a source and a destination endpoint. The source is the initial sender of the message and the destination is the ultimate receiver, as shown in Figure 18.1, “Web Services Reliable Messaging”.

Figure 18.1. Web Services Reliable Messaging



The flow of WS-RM messages can be described as follows:

1. The RM source sends a **`CreateSequence`** protocol message to the RM destination. This contains a reference for the endpoint that receives acknowledgements (the **`wsrm:AcksTo`** endpoint).

2. The RM destination sends a **`CreateSequenceResponse`** protocol message back to the RM source. This message contains the sequence ID for the RM sequence session.

3. The RM source adds an RM **Sequence** header to each message sent by the application source. This header contains the sequence ID and a unique message ID.

4. The RM source transmits each message to the RM destination.

5. The RM destination acknowledges the receipt of the message from the RM source by sending messages that contain the RM **SequenceAcknowledgement** header.

6. The RM destination delivers the message to the application destination in an exactly-once-in-order fashion.

7. The RM source retransmits a message that it has not yet received an acknowledgement.

   The first retransmission attempt is made after a base retransmission interval. Successive retransmission attempts are made, by default, at exponential back-off intervals or, alternatively, at fixed intervals. For more details, see Section 18.4, “Configuring WS-RM”.

This entire process occurs symmetrically for both the request and the response message; that is, in the case of the response message, the server acts as the RM source and the client acts as the RM destination.

WS-RM delivery assurances

WS-RM guarantees reliable message delivery in a distributed environment, regardless of the transport protocol used. Either the source or the destination endpoint logs an error if reliable delivery can not be assured.

Supported specifications

Apache CXF supports the 2005/02 version of the WS-RM specification, which is based on the WS-Addressing 2004/08 specification.

Further information

For detailed information on WS-RM, see the specification at http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf.

18.2. WS-RM INTERCEPTORS

Overview

In Apache CXF, WS-RM functionality is implemented as interceptors. The Apache CXF runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the application's interceptor chain includes the WS-RM interceptors, the application can participate in reliable messaging sessions. The WS-RM interceptors handle the collection and aggregation of the message chunks. They also handle all of the acknowledgement and retransmission logic.

Apache CXF WS-RM Interceptors

The Apache CXF WS-RM implementation consists of four interceptors, which are described in Table 18.1, “Apache CXF WS-ReliableMessaging Interceptors”.

Table 18.1. Apache CXF WS-ReliableMessaging Interceptors

Interceptor	Description
`org.apache.cxf.ws.rm.RMOutInterceptor`	Deals with the logical aspects of providing reliability guarantees for outgoing messages. Responsible for sending the `CreateSequence` requests and waiting for their `CreateSequenceResponse` responses. Also responsible for aggregating the sequence properties—ID and message number—for an application message.
`org.apache.cxf.ws.rm.RMInInterceptor`	Responsible for intercepting and processing RM protocol messages and `SequenceAcknowledgement` messages that are piggybacked on application messages.
`org.apache.cxf.ws.rm.soap.RMSoapInterceptor`	Responsible for encoding and decoding the reliability properties as SOAP headers.
`org.apache.cxf.ws.rm.RetransmissionInterceptor`	Responsible for creating copies of application messages for future resending.

Enabling WS-RM

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, when intercepting the first application message on the outbound interceptor chain, the `RMOutInterceptor` sends a `CreateSequence` request and waits to process the original application message until it receives the `CreateSequenceResponse` response. In addition, the WS-RM interceptors add the sequence headers to the application messages and, on the destination side, extract them from the messages. It is not necessary to make any changes to your application code to make the exchange of messages reliable.

For more information on how to enable WS-RM, see Section 18.3, “Enabling WS-RM”.

Configuring WS-RM Attributes

You control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default Apache CXF attempts to maximize the lifetime of a sequence, thus reducing the overhead incurred by the out-of-band WS-RM protocol messages. To enforce the use of a separate sequence per application message configure the WS-RM source’s sequence termination policy (setting the maximum sequence length to `1`).

For more information on configuring WS-RM behavior, see Section 18.4, “Configuring WS-RM”.

18.3. ENABLING WS-RM

Overview

To enable reliable messaging, the WS-RM interceptors must be added to the interceptor chains for both inbound and outbound messages and faults. Because the WS-RM interceptors use WS-Addressing, the WS-Addressing interceptors must also be present on the interceptor chains.

You can ensure the presence of these interceptors in one of two ways:

- Explicitly, by adding them to the dispatch chains using Spring beans
- Implicitly, using WS-Policy assertions, which cause the Apache CXF runtime to transparently add the interceptors on your behalf.

Spring beans—explicitly adding interceptors

To enable WS-RM add the WS-RM and WS-Addressing interceptors to the Apache CXF bus, or to a consumer or service endpoint using Spring bean configuration. This is the approach taken in the WS-RM sample that is found in the ***InstallDir*/samples/ws_rm** directory. The configuration file, **ws-rm.cxf**, shows the WS-RM and WS-Addressing interceptors being added one-by-one as Spring beans (see Example 18.1, "Enabling WS-RM Using Spring Beans").

Example 18.1. Enabling WS-RM Using Spring Beans

```
<?xml version="1.0" encoding="UTF-8"?>
1 <beans xmlns="http://www.springframework.org/schema/beans"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
   beans http://www.springframework.org/schema/beans/spring-beans.xsd">
2    <bean id="mapAggregator"
  class="org.apache.cxf.ws.addressing.MAPAggregator"/>
   <bean id="mapCodec"
class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
3    <bean id="rmLogicalOut"
  class="org.apache.cxf.ws.rm.RMOutInterceptor">
       <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmLogicalIn" class="org.apache.cxf.ws.rm.RMInInterceptor">
       <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmCodec"
class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>
   <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
4        <property name="inInterceptors">
             <list>
               <ref bean="mapAggregator"/>
               <ref bean="mapCodec"/>
               <ref bean="rmLogicalIn"/>
               <ref bean="rmCodec"/>
           </list>
       </property>
5        <property name="inFaultInterceptors">
             <list>
               <ref bean="mapAggregator"/>
               <ref bean="mapCodec"/>
               <ref bean="rmLogicalIn"/>
               <ref bean="rmCodec"/>
           </list>
       </property>
```

```
6         <property name="outInterceptors">
              <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
7         <property name="outFaultInterceptors">
              <list>
                <ref bean="mapAggregator">
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
    </bean>
</beans>
```

The code shown in Example 18.1, "Enabling WS-RM Using Spring Beans" can be explained as follows:

1. A Apache CXF configuration file is a Spring XML file. You must include an opening Spring **beans** element that declares the namespaces and schema files for the child elements that are encapsulated by the **beans** element.
2. Configures each of the WS-Addressing interceptors—**`MAPAggregator`** and **`MAPCodec`**. For more information on WS-Addressing, see Chapter 17, *Deploying WS-Addressing*.
3. Configures each of the WS-RM interceptors—**`RMOutInterceptor`**, **`RMInInterceptor`**, and **`RMSoapInterceptor`**.
4. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound messages.
5. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound faults.
6. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound messages.
7. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound faults.

WS-Policy framework—implicitly adding interceptors

The WS-Policy framework provides the infrastructure and APIs that allow you to use WS-Policy. It is compliant with the November 2006 draft publications of the Web Services Policy 1.5—Framework and Web Services Policy 1.5—Attachment specifications.

To enable WS-RM using the Apache CXF WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint. Example 18.2, "Configuring WS-RM using WS-Policy" shows a reference bean nested within a **`jaxws:feature`** element. The reference bean specifies the **`AddressingPolicy`**, which is defined as a separate element within the same configuration file.

Example 18.2. Configuring WS-RM using WS-Policy

```
<jaxws:client>
    <jaxws:features>
      <ref bean="AddressingPolicy"/>
    </jaxws:features>
</jaxws:client>
<wsp:Policy wsu:Id="AddressingPolicy"
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
    <wsam:Addressing>
      <wsp:Policy>
        <wsam:NonAnonymousResponses/>
      </wsp:Policy>
    </wsam:Addressing>
</wsp:Policy>
```

2. Add a reliable messaging policy to the **wsdl:service** element—or any other WSDL element that can be used as an attachment point for policy or policy reference elements—to your WSDL file, as shown in Example 18.3, "Adding an RM Policy to Your WSDL File" .

Example 18.3. Adding an RM Policy to Your WSDL File

```
<wsp:Policy wsu:Id="RM"
   xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd">
    <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
    </wsam:Addressing>
    <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
    </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
    <wsdl:port binding="tns:GreeterSOAPBinding"
name="GreeterPort">
        <soap:address
location="http://localhost:9020/SoapContext/GreeterPort"/>
        <wsp:PolicyReference URI="#RM"
xmlns:wsp="http://www.w3.org/2006/07/ws-policy"/>
    </wsdl:port>
</wsdl:service>
```

18.4. CONFIGURING WS-RM

You can configure WS-RM by:

- Setting Apache CXF-specific attributes that are defined in the Apache CXF WS-RM manager namespace, **http://cxf.apache.org/ws/rm/manager**.

- Setting standard WS-RM policy attributes that are defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace.

18.4.1. Configuring Apache CXF-Specific WS-RM Attributes

Overview

To configure the Apache CXF-specific attributes, use the **rmManager** Spring bean. Add the following to your configuration file:

- The **http://cxf.apache.org/ws/rm/manager** namespace to your list of namespaces.
- An **rmManager** Spring bean for the specific attribute that your want to configure.

Example 18.4, "Configuring Apache CXF-Specific WS-RM Attributes" shows a simple example.

Example 18.4. Configuring Apache CXF-Specific WS-RM Attributes

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager
http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd">
...
<wsrm-mgr:rmManager>
<!--
  ...Your configuration goes here
-->
</wsrm-mgr:rmManager>
```

Children of the rmManager Spring bean

Table 18.2, "Children of the rmManager Spring Bean" shows the child elements of the **rmManager** Spring bean, defined in the **http://cxf.apache.org/ws/rm/manager** namespace.

Table 18.2. Children of the rmManager Spring Bean

Element	Description
RMAssertion	An element of type RMAssertion
deliveryAssurance	An element of type DeliveryAssuranceType that describes the delivery assurance that should apply
sourcePolicy	An element of type SourcePolicyType that allows you to configure details of the RM source

Element	Description
`destinationPolicy`	An element of type DestinationPolicyType that allows you to configure details of the RM destination

Example

For an example, see the section called “Maximum unacknowledged messages threshold” .

18.4.2. Configuring Standard WS-RM Policy Attributes

Overview

You can configure standard WS-RM policy attributes in one of the following ways:

- RMAssertion in rmManager Spring bean
- Policy within a feature
- WSDL file
- External attachment

WS-Policy RMAssertion Children

Table 18.3, “Children of the WS-Policy RMAssertion Element” shows the elements defined in the `http://schemas.xmlsoap.org/ws/2005/02/rm/policy` namespace:

Table 18.3. Children of the WS-Policy RMAssertion Element

Name	Description
`InactivityTimeout`	Specifies the amount of time that must pass without receiving a message before an endpoint can consider an RM sequence to have been terminated due to inactivity.
`BaseRetransmissionInterval`	Sets the interval within which an acknowledgement must be received by the RM Source for a given message. If an acknowledgement is not received within the time set by the `BaseRetransmissionInterval`, the RM Source will retransmit the message.
`ExponentialBackoff`	Indicates the retransmission interval will be adjusted using the commonly known exponential backoff algorithm (Tanenbaum). For more information, see *Computer Networks*, Andrew S. Tanenbaum, Prentice Hall PTR, 2003.

Name	Description
AcknowledgementInterval	In WS-RM, acknowledgements are sent on return messages or sent stand-alone. If a return message is not available to send an acknowledgement, an RM Destination can wait for up to the acknowledgement interval before sending a stand-alone acknowledgement. If there are no unacknowledged messages, the RM Destination can choose not to send an acknowledgement.

More detailed reference information

For more detailed reference information, including descriptions of each element's sub-elements and attributes, please refer to http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd.

RMAssertion in rmManager Spring bean

You can configure standard WS-RM policy attributes by adding an **RMAssertion** within a Apache CXF **rmManager** Spring bean. This is the best approach if you want to keep all of your WS-RM configuration in the same configuration file; that is, if you want to configure Apache CXF-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in Example 18.5, "Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean" shows:

- A standard WS-RM policy attribute, **BaseRetransmissionInterval**, configured using an **RMAssertion** within an **rmManager** Spring bean.
- An Apache CXF-specific RM attribute, **intraMessageThreshold**, configured in the same configuration file.

Example 18.5. Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
    <wsrm-mgr:destinationPolicy>
        <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
    </wsrm-mgr:destinationPolicy>
</wsrm-mgr:rmManager>
</beans>
```

Policy within a feature

You can configure standard WS-RM policy attributes within features, as shown in Example 18.6, "Configuring WS-RM Attributes as a Policy within a Feature".

Example 18.6. Configuring WS-RM Attributes as a Policy within a Feature

```
<xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:wsa="http://cxf.apache.org/ws/addressing"
        xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd"
        xmlns:jaxws="http://cxf.apache.org/jaxws"
        xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-
policy.xsd
http://cxf.apache.org/ws/addressing
http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <jaxws:endpoint name="
{http://cxf.apache.org/greeter_control}GreeterPort"
createdFromAPI="true">
        <jaxws:features>
               <wsp:Policy>
                   <wsrm:RMAssertion
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                     <wsrm:AcknowledgementInterval Milliseconds="200"
/>
                   </wsrm:RMAssertion>
                   <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                       <wsp:Policy>
                            <wsam:NonAnonymousResponses/>
                       </wsp:Policy>
                   </wsam:Addressing>
              </wsp:Policy>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

WSDL file

If you use the WS-Policy framework to enable WS-RM, you can configure standard WS-RM policy attributes in a WSDL file. This is a good approach if you want your service to interoperate and use WS-RM seamlessly with consumers deployed to other policy-aware Web services stacks.

For an example, see the section called "WS-Policy framework—implicitly adding interceptors" where the base retransmission interval is configured in the WSDL file.

External attachment

You can configure standard WS-RM policy attributes in an external attachment file. This is a good approach if you cannot, or do not want to, change your WSDL file.

Example 18.7, "Configuring WS-RM in an External Attachment" shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 18.7. Configuring WS-RM in an External Attachment

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsp:PolicyAttachment>
        <wsp:AppliesTo>
           <wsa:EndpointReference>

<wsa:Address>http://localhost:9020/SoapContext/GreeterPort</wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:Policy>
            <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                <wsp:Policy/>
            </wsam:Addressing>
            <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                <wsrmp:BaseRetransmissionInterval
Milliseconds="30000"/>
            </wsrmp:RMAssertion>
        </wsp:Policy>
    </wsp:PolicyAttachment>
</attachments>/
```

18.4.3. WS-RM Configuration Use Cases

Overview

This subsection focuses on configuring WS-RM attributes from a use case point of view. Where an attribute is a standard WS-RM policy attribute, defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace, only the example of setting it in an **`RMAssertion`** within an **`rmManager`** Spring bean is shown. For details of how to set such attributes as a policy within a feature; in a WSDL file, or in an external attachment, see Section 18.4.2, "Configuring Standard WS-RM Policy Attributes".

The following use cases are covered:

- Base retransmission interval
- Exponential backoff for retransmission
- Acknowledgement interval
- Maximum unacknowledged messages threshold
- Maximum length of an RM sequence

- Message delivery assurance policies

Base retransmission interval

The **BaseRetransmissionInterval** element specifies the interval at which an RM source retransmits a message that has not yet been acknowledged. It is defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd schema file. The default value is 3000 milliseconds.

Example 18.8, "Setting the WS-RM Base Retransmission Interval" shows how to set the WS-RM base retransmission interval.

Example 18.8. Setting the WS-RM Base Retransmission Interval

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Exponential backoff for retransmission

The **ExponentialBackoff** element determines if successive retransmission attempts for an unacknowledged message are performed at exponential intervals.

The presence of the **ExponentialBackoff** element enables this feature. An exponential backoff ratio of **2** is used by default.

Example 18.9, "Setting the WS-RM Exponential Backoff Property" shows how to set the WS-RM exponential backoff for retransmission.

Example 18.9. Setting the WS-RM Exponential Backoff Property

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:ExponentialBackoff="4"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Acknowledgement interval

The **AcknowledgementInterval** element specifies the interval at which the WS-RM destination sends asynchronous acknowledgements. These are in addition to the synchronous acknowledgements

that it sends on receipt of an incoming message. The default asynchronous acknowledgement interval is **0** milliseconds. This means that if the **AcknowledgementInterval** is not configured to a specific value, acknowledgements are sent immediately (that is, at the first available opportunity).

Asynchronous acknowledgements are sent by the RM destination only if both of the following conditions are met:

- The RM destination is using a non-anonymous **wsrm:acksTo** endpoint.
- The opportunity to piggyback an acknowledgement on a response message does not occur before the expiry of the acknowledgement interval.

Example 18.10, "Setting the WS-RM Acknowledgement Interval" shows how to set the WS-RM acknowledgement interval.

Example 18.10. Setting the WS-RM Acknowledgement Interval

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Maximum unacknowledged messages threshold

The **maxUnacknowledged** attribute sets the maximum number of unacknowledged messages that can accrue per sequence before the sequence is terminated.

Example 18.11, "Setting the WS-RM Maximum Unacknowledged Message Threshold" shows how to set the WS-RM maximum unacknowledged messages threshold.

Example 18.11. Setting the WS-RM Maximum Unacknowledged Message Threshold

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Maximum length of an RM sequence

The **maxLength** attribute sets the maximum length of a WS-RM sequence. The default value is **0**, which means that the length of a WS-RM sequence is unbound.

When this attribute is set, the RM endpoint creates a new RM sequence when the limit is reached, and after receiving all of the acknowledgements for the previously sent messages. The new message is sent using a newsequence.

Example 18.12, "Setting the Maximum Length of a WS-RM Message Sequence" shows how to set the maximum length of an RM sequence.

Example 18.12. Setting the Maximum Length of a WS-RM Message Sequence

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Message delivery assurance policies

You can configure the RM destination to use the following delivery assurance policies:

- **`AtMostOnce`** — The RM destination delivers the messages to the application destination only once. If a message is delivered more than once an error is raised. It is possible that some messages in a sequence may not be delivered.
- **`AtLeastOnce`** — The RM destination delivers the messages to the application destination at least once. Every message sent will be delivered or an error will be raised. Some messages might be delivered more than once.
- **`InOrder`** — The RM destination delivers the messages to the application destination in the order that they are sent. This delivery assurance can be combined with the **`AtMostOnce`** or **`AtLeastOnce`** assurances.

Example 18.13, "Setting the WS-RM Message Delivery Assurance Policy" shows how to set the WS-RM message delivery assurance.

Example 18.13. Setting the WS-RM Message Delivery Assurance Policy

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:deliveryAssurance>
        <wsrm-mgr:AtLeastOnce />
    </wsrm-mgr:deliveryAssurance>
</wsrm-mgr:reliableMessaging>
</beans>
```

18.5. CONFIGURING WS-RM PERSISTENCE

Overview

The Apache CXF WS-RM features already described in this chapter provide reliability for cases such as network failures. WS-RM persistence provides reliability across other types of failure such as an RM source or an RM destination crash.

WS-RM persistence involves storing the state of the various RM endpoints in persistent storage. This enables the endpoints to continue sending and receiving messages when they are reincarnated.

Apache CXF enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, Apache CXF includes Derby for out-of-the-box deployment. In addition, the persistent store is also exposed using a Java API.

IMPORTANT

WS-RM persistence is supported for oneway calls only, and it is disabled by default.

How it works

Apache CXF WS-RM persistence works as follows:

- At the RM source endpoint, an outgoing message is persisted before transmission. It is evicted from the persistent store after the acknowledgement is received.
- After a recovery from crash, it recovers the persisted messages and retransmits until all the messages have been acknowledged. At that point, the RM sequence is closed.
- At the RM destination endpoint, an incoming message is persisted, and upon a successful store, the acknowledgement is sent. When a message is successfully dispatched, it is evicted from the persistent store.
- After a recovery from a crash, it recovers the persisted messages and dispatches them. It also brings the RM sequence to a state where new messages are accepted, acknowledged, and delivered.

Enabling WS-persistence

To enable WS-RM persistence, you must specify the object implementing the persistent store for WS-RM. You can develop your own or you can use the JDBC based store that comes with Apache CXF.

The configuration shown in Example 18.14, “Configuration for the Default WS-RM Persistence Store” enables the JDBC-based store that comes with Apache CXF.

Example 18.14. Configuration for the Default WS-RM Persistence Store

```
<bean id="RMTxStore"
class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore"/>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <property name="store" ref="RMTxStore"/>
</wsrm-mgr:rmManager>
```

Configuring WS-persistence

The JDBC-based store that comes with Apache CXF supports the properties shown in Table 18.4, "JDBC Store Properties".

Table 18.4. JDBC Store Properties

Attribute Name	Type	Default Setting
driverClassName	String	`org.apache.derby.jdbc.EmbeddedDriver`
userName	String	null
passWord	String	null
url	String	jdbc:derby:rmdb;create=true

The configuration shown in Example 18.15, "Configuring the JDBC Store for WS-RM Persistence" enables the JDBC-based store that comes with Apache CXF, while setting the driverClassName and url to non-default values.

Example 18.15. Configuring the JDBC Store for WS-RM Persistence

```
<bean id="RMTxStore"
class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore">
    <property name="driverClassName" value="com.acme.jdbc.Driver"/>
    <property name="url" value="jdbc:acme:rmdb;create=true"/>
</bean>
```

CHAPTER 19. ENABLING HIGH AVAILABILITY

Abstract

This chapter explains how to enable and configure high availability in the Apache CXF runtime.

19.1. INTRODUCTION TO HIGH AVAILABILITY

Overview

Scalable and reliable applications require high availability to avoid any single point of failure in a distributed system. You can protect your system from single points of failure using *replicated services*.

A replicated service is comprised of multiple instances, or *replicas*, of the same service. Together these act as a single logical service. Clients invoke requests on the replicated service, and Apache CXF delivers the requests to one of the member replicas. The routing to a replica is transparent to the client.

HA with static failover

Apache CXF supports high availability (HA) with static failover in which replica details are encoded in the service WSDL file. The WSDL file contains multiple ports, and can contain multiple hosts, for the same service. The number of replicas in the cluster remains static as long as the WSDL file remains unchanged. Changing the cluster size involves editing the WSDL file.

19.2. ENABLING HA WITH STATIC FAILOVER

Overview

To enable HA with static failover, you must do the following:

1. Encode replica details in your service WSDL file
2. Add the clustering feature to your client configuration

Encode replica details in your service WSDL file

You must encode the details of the replicas in your cluster in your service WSDL file. Example 19.1, "Enabling HA with Static Failover—WSDL File" shows a WSDL file extract that defines a service cluster of three replicas.

Example 19.1. Enabling HA with Static Failover—WSDL File

```
1 <wsdl:service name="ClusteredService">
2     <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica1">
          <soap:address
location="http://localhost:9001/SoapContext/Replica1"/>
    </wsdl:port>

3     <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica2">
          <soap:address
```

```
location="http://localhost:9002/SoapContext/Replica2"/>
    </wsdl:port>

4     <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica3">
          <soap:address
location="http://localhost:9003/SoapContext/Replica3"/>
    </wsdl:port>

</wsdl:service>
```

The WSDL extract shown in Example 19.1, "Enabling HA with Static Failover—WSDL File" can be explained as follows:

1. Defines a service, **ClusterService**, which is exposed on three ports:
    1. **Replica1**
    2. **Replica2**
    3. **Replica3**
2. Defines **Replica1** to expose the **ClusterService** as a SOAP over HTTP endpoint on port **9001**.
3. Defines **Replica2** to expose the **ClusterService** as a SOAP over HTTP endpoint on port **9002**.
4. Defines **Replica3** to expose the **ClusterService** as a SOAP over HTTP endpoint on port **9003**.

Add the clustering feature to your client configuration

In your client configuration file, add the clustering feature as shown in Example 19.2, "Enabling HA with Static Failover—Client Configuration".

Example 19.2. Enabling HA with Static Failover—Client Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:clustering="http://cxf.apache.org/clustering"
         xsi:schemaLocation="http://cxf.apache.org/jaxws
         http://cxf.apache.org/schemas/jaxws.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:client name="
{http://apache.org/hello_world_soap_http}Replica1"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="
```

```
{http://apache.org/hello_world_soap_http}Replica2"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="
{http://apache.org/hello_world_soap_http}Replica3"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

</beans>
```

19.3. CONFIGURING HA WITH STATIC FAILOVER

Overview

By default, HA with static failover uses a sequential strategy when selecting a replica service if the original service with which a client is communicating becomes unavailable, or fails. The sequential strategy selects a replica service in the same sequential order every time it is used. Selection is determined by Apache CXF's internal service model and results in a deterministic failover pattern.

Configuring a random strategy

You can configure HA with static failover to use a random strategy instead of the sequential strategy when selecting a replica. The random strategy selects a random replica service each time a service becomes unavailable, or fails. The choice of failover target from the surviving members in a cluster is entirely random.

To configure the random strategy, add the configuration shown in Example 19.3, "Configuring a Random Strategy for Static Failover" to your client configuration file.

Example 19.3. Configuring a Random Strategy for Static Failover

```
<beans ...>
1     <bean id="Random"
  class="org.apache.cxf.clustering.RandomStrategy"/>

    <jaxws:client name="
{http://apache.org/hello_world_soap_http}Replica3"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover>
2                 <clustering:strategy>
                      <ref bean="Random"/>
                  </clustering:strategy>
            </clustering:failover>
        </jaxws:features>
    </jaxws:client>
```

```
</beans>
```

The configuration shown in Example 19.3, “Configuring a Random Strategy for Static Failover” can be explained as follows:

1. Defines a **Random** bean and implementation class that implements the random strategy.
2. Specifies that the random strategy is used when selecting a replica.

CHAPTER 20. ENABLING HIGH AVAILABILITY IN FUSE FABRIC

Abstract

When all of your servers and clients are deployed within the same fabric, you can use an alternative mechanism for implementing high availability cluster, which works by exploiting the fabric registry. Because all the parts of the application must be deployed on the same fabric, this mechanism is suitable for deployment on a LAN.

20.1. LOAD BALANCING CLUSTER

20.1.1. Introduction to Load Balancing

Overview

The fabric load balancing mechanism exploits the fact that fabric provides a distributed fabric registry, which is accessible to all of the container in the fabric. This makes it possible to use the fabric registry as a discovery mechanism for locating WS endpoints in the fabric. By storing all of the endpoint addresses belonging to a particular cluster under the same registry node, any WS clients in the fabric can easily discover the location of the endpoints in the cluster.

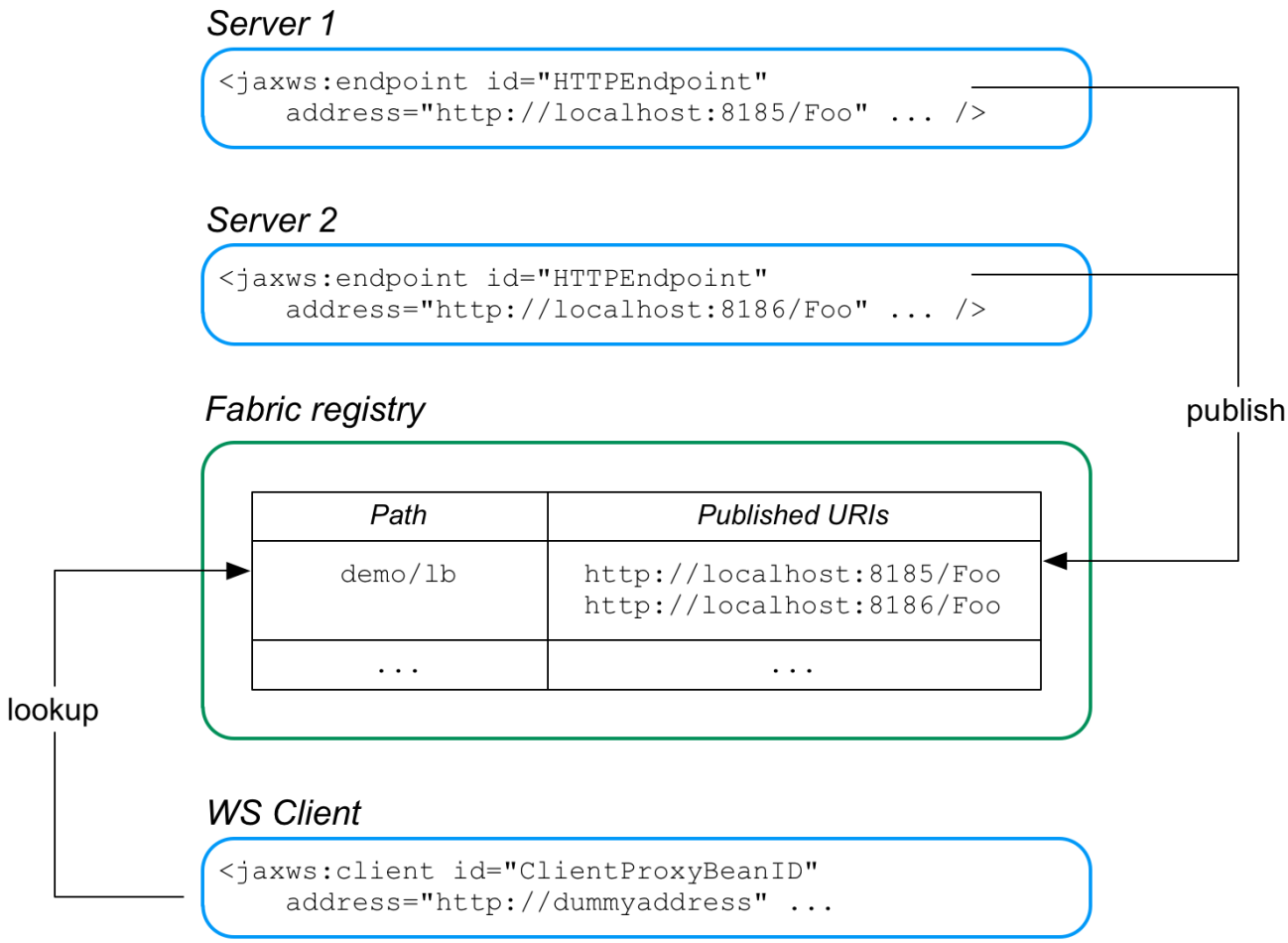
Fuse Fabric

A fabric is a distributed collection of containers that share a common database of configuration settings (the *fabric registry*). Every container in the fabric has a fabric agent deployed in it, which manages the container and redeploys applications to the container whenever a new profile is assigned to the container (a profile is the basic deployment unit in a fabric).

Load-balancing cluster

Figure 20.1, “Fabric Load Balancing for Apache CXF” gives an overview of the fabric load balancing mechanism for Apache CXF endpoints.

Figure 20.1. Fabric Load Balancing for Apache CXF



In this example, two WS servers are created, with the URIs, **`http://localhost:8185/Foo`** and **`http://localhost:8186/Foo`**. For both of these servers, the load balancer feature is configured to store the cluster endpoints under the path, **`demo/lb`**, in the fabric registry.

Now, when the WS client starts, it is configured to look up the cluster path, **`demo/lb`**, in the fabric registry. Because the **`demo/lb`** path is associated with multiple endpoint addresses, fabric implements a random load balancing algorithm to choose one of the available URIs to connect to.

FabricLoadBalancerFeature

The fabric load balancer feature is implemented by the following class:

```
io.fabric8.cxf.FabricLoadBalancerFeature
```

The **`FabricLoadBalancerFeature`** class exposes the following bean properties:

`fabricPath`

This property specifies a node in the fabric registry (specified relative to the base node, **`/fabric/cxf/endpoints`**) that is used to store the data for a particular endpoint cluster.

`curator`

A proxy reference to the OSGi service exposed by the fabric agent (of type, **`org.apache.curator.framework.CuratorFramework`**).

`maximumConnectionTimeout`

The maximum length of time to attempt to connect to the fabric agent, specified in milliseconds. The default is 10000 (10 seconds).

`connectionRetryTime`

How long to wait between connection attempts, specified in milliseconds. The default is 100.

`loadBalanceStrategy`

By implementing a bean of type **`io.fabric8.cxf.LoadBalanceStrategy`** and setting this property, you can customise the load balancing algorithm used by the load balancing feature.

Prerequisites

To use the fabric load balancer feature in your application, your project must satisfy the following prerequisites:

- the section called "Maven dependency".
- the section called "OSGi package import".
- the section called "Fabric deployment".
- the section called "Required feature".

Maven dependency

The fabric load balancer feature requires the **`fabric-cxf`** Maven artifact. Add the following dependency to your project's POM file:

```
<dependency>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric-cxf</artifactId>
    <version>6.1.0.redhat-379</version>
</dependency>
```

OSGi package import

If you are packaging your project as an OSGi bundle, you must add **`io.fabric8.cxf`** to the list of imported packages. For example, using the Maven bundle plug-in, you can specify this package import by adding **`io.fabric8.cxf`** to the comma-separated list in the **`Import-Package`** element, as follows:

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>2.2.0</version>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>${project.artifactId}</Bundle-
SymbolicName>
            <Import-Package>
```

```
                ...
 io.fabric8.cxf,
                *
            </Import-Package>
            ...
        </instructions>
    </configuration>
</plugin>
```

Fabric deployment

When you come to deploy your application into a Red Hat JBoss Fuse container, you *must* deploy it into a fabric. The fabric load balancer feature is not supported in a standalone container.

Required feature

The fabric load balancer requires the **`fabric-cxf`** Apache Karaf feature to be installed in the container. In the context of a fabric, this means you must add the **`fabric-cxf`** feature to the relevant deployment profile. For example, if you are using the **`cxf-lb-server`** profile to deploy a load-balancing WS server, you can add the **`fabric-cxf`** feature by entering the following console command:

```
JBossFuse:karaf@root> profile-edit -f fabric-cxf cxf-lb-server
```

20.1.2. Configure the Server

Overview

To configure a WS server to use fabric load balancing, you must configure a fabric load balancer feature and install it in the default Apache CXF bus instance. This section describes how to configure the load balancer feature in Spring XML and in blueprint XML.

Prerequisites

For the basic prerequisites to build a fabric load-balancing WS server, see the section called "Prerequisites".

Blueprint XML

The following fragment from a blueprint XML file shows how to add the fabric load balancer feature, **`FabricLoadBalancerFeature`**, to an Apache CXF bus. Any Apache CXF endpoints subsequently created on this bus will automatically have the load-balancer feature enabled.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           ...
           xmlns:cxf="http://cxf.apache.org/blueprint/core"
           ...
>
    ...
    <reference id="curator"
               interface="org.apache.curator.framework.CuratorFramework"
/>
```

```
    <!-- The FabricFailOverFeature will try to access other service
endpoint with round rad -->
    <bean id="fabricLoadBalancerFeature"
class="io.fabric8.cxf.FabricLoadBalancerFeature">
        <property name="curator" ref="curator" />
        <property name="fabricPath" value="cxf/demo" />
    </bean>

    <!-- setup the feature on the bus to help publish the services to the
fabric-->
    <cxf:bus bus="cxf">
        <cxf:features>
            <ref component-id="fabricLoadBalancerFeature"/>
        </cxf:features>
    </cxf:bus>
    ...
</blueprint>
```

The following beans are used to install the fabric load-balancer feature:

`curator` reference

The **`curator`** reference is a proxy of the local fabric agent, which it accesses through the **`org.apache.curator.framework.CuratorFramework`** interface. This reference is needed in order to integrate the load balancer feature with the underlying fabric.

`FabricLoadBalancerFeature` bean

The **`FabricLoadBalancerFeature`** bean is initialized with the following properties:

`curator`

A reference to the Apache Curator client, **`CuratorFramework`**.

`fabricPath`

The path of a node in the fabric registry, where the cluster data is stored (for example, the addresses of the endpoints in the load-balancing cluster). The node path is specified relative to the base node, **`/fabric/cxf/endpoints`**.

Apache CXF bus

The **`cxf:bus`** element installs the fabric load balancer feature in the default bus instance.

Example using Blueprint XML

Example 20.1, “WS Server with Fabric Load Balancer Feature” shows a complete Blueprint XML example of a WS endpoint configured to use the fabric load balancing feature.

Example 20.1. WS Server with Fabric Load Balancer Feature

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cxf="http://cxf.apache.org/blueprint/core"
           xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
```

```
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.0.0"
           xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
           http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
           http://cxf.apache.org/schemas/blueprint/core
           http://cxf.apache.org/schemas/blueprint/core.xsd
           http://cxf.apache.org/blueprint/jaxws
           http://cxf.apache.org/blueprint/jaxws.xsd
           ">

    <reference id="curator"
interface="org.apache.curator.framework.CuratorFramework" />

    <!-- The FabricFailOverFeature will try to access other service
endpoint with round rad -->
    <bean id="fabicLoadBalancerFeature"
class="io.fabric8.cxf.FabricLoadBalancerFeature">
        <property name="curator" ref="curator" />
        <property name="fabricPath" value="cxf/demo" />
    </bean>

    <!-- setup the feature on the bus to help publish the services to
the fabric-->
    <cxf:bus bus="cxf">
        <cxf:features>
            <ref component-id="fabicLoadBalancerFeature"/>
        </cxf:features>
    </cxf:bus>

    <bean id="hello1" class="io.fabric8.demo.cxf.server.HelloImpl">
        <property name="hello" value="Hi"/>
    </bean>

    <bean id="hello2" class="io.fabric8.demo.cxf.server.HelloImpl">
        <property name="hello" value="Hello"/>
    </bean>

    <!--
    TODO: We should use address in the form of
http://$[bind.address]:$[app1.port]/server/server1, but currently only
fuseenterprise
    has appX.port system properties defined
    -->

    <!-- publish the service with the address of fail, cxf client will
get the simulated IOException -->
    <jaxws:server id="service1" serviceClass="io.fabric8.demo.cxf.Hello"
address="http://localhost:9000/server/server1">
        <jaxws:serviceBean>
            <ref component-id="hello1" />
        </jaxws:serviceBean>
    </jaxws:server>

    <jaxws:server id="service2" serviceClass="io.fabric8.demo.cxf.Hello"
address="http://localhost:9000/server/server2">
```

```
        <jaxws:serviceBean>
            <ref component-id="hello2" />
        </jaxws:serviceBean>
    </jaxws:server>

</blueprint>
```

The preceding Spring XML configuration consists of the following main sections:

- *Enabling the fabric load balancing feature*—the fabric load balancing feature is installed in the default bus instance, as previously described. In this example, the **`fabricPath`** property is set to the value, **`cxf/demo`**.
- *Creating the WS endpoints*—create the WS endpoints in the usual way, using the **`jaxws:server`** element (this can be used as an alternative to the **`jaxws:endpoint`** element). By default, this endpoint is *automatically* associated with the default bus instance, which has load balancing enabled.

20.1.3. Configure the Client

Overview

To configure a WS client to use fabric load balancing, you must install the fabric load balancer feature directly in the client proxy instance. This section describes how to configure the load balancer feature in Blueprint XML, and by programming in Java.

Prerequisites

For the basic prerequisites to build a fabric load-balancing WS client, see the section called "Prerequisites".

Blueprint XML

The following fragment from a blueprint XML file shows how to add the fabric load balancer feature, **`FabricLoadBalancerFeature`**, directly into a WS client proxy instance.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           ...
           xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
           xmlns:cxf="http://cxf.apache.org/blueprint/core"
           ...
>
    <!-- Create a client proxy, with load balancing enabled -->
    <jaxws:client id="ClientProxyBeanID"
        address="http://dummyaddress"
        serviceClass="SEI">
        <jaxws:features>
            <ref component-id="fabricLoadBalancerFeature" />
        </jaxws:features>
    </jaxws:client>
    ...
    <reference id="curator"
               interface="org.apache.curator.framework.CuratorFramework"
```

```
/>

    <!-- The FabricFailOverFeature will try to access other service
endpoint with round rad -->
    <bean id="fabricLoadBalancerFeature"
class="io.fabric8.cxf.FabricLoadBalancerFeature">
        <property name="curator" ref="curator" />
        <property name="fabricPath" value="ZKPath" />
    </bean>
    ...
</blueprint>
```

The fabric load balancer feature is installed directly into the WS client proxy by inserting it as a child of the **jaxws:features** element (or, as in this case, by inserting a bean reference to the actual instance). The following beans are used to initialise the fabric load-balancer feature:

curator reference

The **curator** reference is a proxy of the local fabric agent, which it accesses through the **org.apache.curator.framework.CuratorFramework** interface. This reference is needed in order to integrate the load balancer feature with the underlying fabric.

FabricLoadBalancerFeature bean

The **FabricLoadBalancerFeature** bean is initialized with the following properties:

curator

A reference to the Apache Curator client, **curator**.

fabricPath

The path of a node in the fabric registry, where the cluster data is stored (for example, the addresses of the endpoints in the load-balancing cluster). The node path is specified relative to the base node, **/fabric/cxf/endpoints**.

Java

As an alternative to using XML configuration, you can enable the fabric load balancing feature on the client side by programming directly in Java. The following example shows how to enable fabric load balancing on a proxy for the **Hello** Web service.

```
// Java
package io.fabric8.demo.cxf.client;

import org.apache.cxf.feature.AbstractFeature;
import org.apache.cxf.frontend.ClientProxyFactoryBean;
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import io.fabric8.cxf.FabricLoadBalancerFeature;
import io.fabric8.demo.cxf.Hello;

import java.util.ArrayList;
import java.util.List;

public class Client {
```

```
    private Hello hello;

    public void initializeHelloProxy() {
        // The feature will try to create a zookeeper client itself
        // by checking the system property of zookeeper.url
        FabricLoadBalancerFeature feature = new
FabricLoadBalancerFeature();
        // Feature will use this path to locate the service
        feature.setFabricPath("demo/lb");

        ClientProxyFactoryBean clientFactory = new
JaxWsProxyFactoryBean();
        clientFactory.setServiceClass(ClientProxyFactoryBean.class);
        // The address is not the actual address that the client will
access
        clientFactory.setAddress("http://dummyaddress");

        List<AbstractFeature> features = new ArrayList<AbstractFeature>();
        features.add(feature);
        // we need to setup the feature on the client factory
        clientFactory.setFeatures(features);

        // Create the proxy of Hello
        hello = clientFactory.create(Hello.class);
    }

    public static void main(String args[]) {
        initializeHelloProxy();
        ...
    }
}
```

In this example, the **fabricPath** property is set to the value, **demo/lb** (which matches the example value used by the server in Example 20.1, "WS Server with Fabric Load Balancer Feature").

The address that the client proxy accesses is set to a dummy value, **http://dummyaddress**, because this value is not used. When the client is initialized, the load balancer feature substitutes the address value retrieved from the **demo/lb** node of the fabric registry.

20.2. FAILOVER CLUSTER

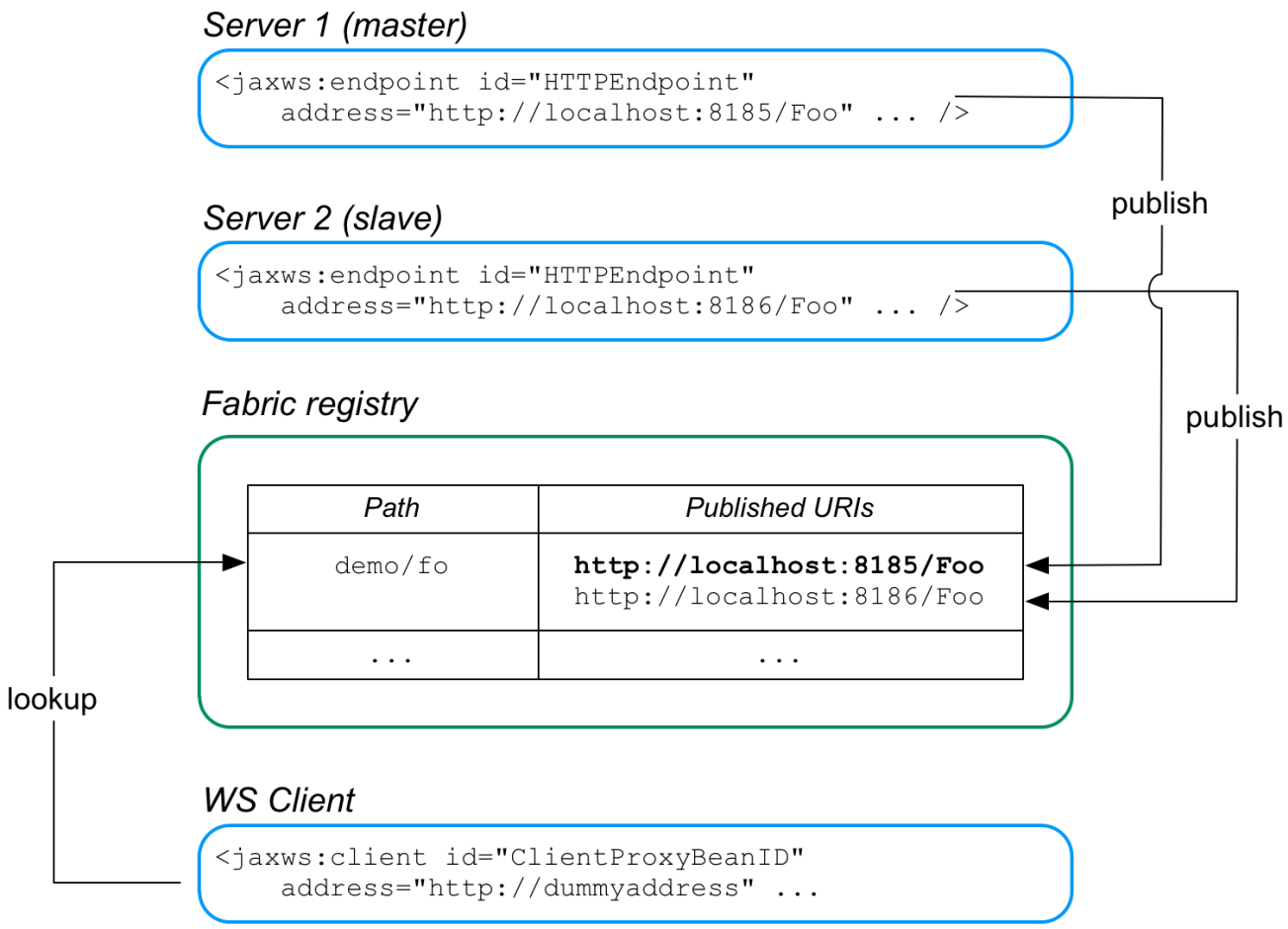
Overview

A failover cluster in Fuse Fabric is based on an ordered list of WS endpoints that are registered under a particular node in the fabric registry. A client detects the failure of a master endpoint by catching the exception that occurs when it tries to make an invocation. When that happens, the client automatically moves to the next available endpoint in the cluster.

Failover cluster

Figure 20.2, "Fabric Failover for Apache CXF" gives an overview of the fabric failover mechanism for Apache CXF endpoints.

Figure 20.2. Fabric Failover for Apache CXF



In this example, two WS servers are created, with the URIs, **`http://localhost:8185/Foo`** and **`http://localhost:8186/Foo`**. In both servers, the failover feature is configured to store the cluster endpoints under the path, **`demo/fo`**, in the fabric registry. The cluster endpoints stored under **`demo/fo`** are *ordered*. The first endpoint in the cluster is the *master* and all of the other endpoints are *slaves*.

The failover algorithm works as follows:

1. When the WS client starts, it is configured to look up the cluster path, **`demo/fo`**, in the fabric registry. The failover feature initially returns the *first* address registered under **`demo/fo`** (the master).

2. At some point, the master server could fail. The client determines whether the master has failed by catching the exception that occurs when it tries to make an invocation: if the caught exception matches one of the exceptions in a specified list (by default, just the **`java.io.IOException`**), the master is deemed to have failed and the client now ignores the corresponding address entry under **`demo/fo`**.

3. The client selects the *next* address entry under **`demo/fo`** and attempts to connect to that server. Assuming that this server is healthy, it is effectively the new master.

4. At some point in the future, if the failed old master is restarted successfully, it creates a new address entry under **`demo/fo`** *after* the existing entries, and is then available to clients, in case the other server (or servers) fail.

FabricFailOverFeature

The fabric failover feature is implemented by the following class:

```
io.fabric8.cxf.FabricFailOverFeature
```

The **FabricFailOverFeature** class exposes the following bean properties:

fabricPath

This property specifies a node in the fabric registry (specified relative to the base node, **/fabric/cxf/endpoints**) that is used to store the data for a particular endpoint cluster.

curator

A proxy reference to the OSGi service (of type, **org.apache.curator.framework.CuratorFramework**) for the Apache Curator client, which is exposed by the fabric agent.

maximumConnectionTimeout

The maximum length of time to attempt to connect to the fabric agent, specified in milliseconds. The default is 10000 (10 seconds).

connectionRetryTime

How long to wait between connection attempts, specified in milliseconds. The default is 100.

exceptions

A semicolon-separated list of exceptions that signal to the client that a server has failed. If not set, this property defaults to **java.io.IOException**.

For example, you could set the **exceptions** property to a value like the following:

```
java.io.IOException;javax.xml.ws.soap.SOAPFaultException
```

Blueprint XML

The configuration of WS servers and WS clients in the failover case is similar to the load balancing case (see Section 20.1.2, “Configure the Server” and Section 20.1.3, “Configure the Client”), except that instead of instantiating and referencing a **FabricLoadBalancerFeature** bean, you must instantiate and reference a **FabricFailOverFeature** bean.

In blueprint XML you can create a **FabricFailOverFeature** bean instance as follows:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           ...
           xmlns:cxf="http://cxf.apache.org/blueprint/core"
           ...
>
    ...
    <!-- Reference the fabric agent -->
    <reference id="curator"
               interface="org.apache.curator.framework.CuratorFramework"
/>
```

```
    <!-- Create the Fabric load balancer feature -->
    <bean id="failoverFeature"
          class="io.fabric8.cxf.FabricFailOverFeature">
        <property name="curator" ref="curator" />
        <property name="fabricPath" value="ZKPath" />
    </bean>
    ...
</blueprint>
```

Remember to customise the value of the **fabricPath** property and to reference the appropriate bean ID (**failoverFeature** in the preceding example).

CHAPTER 21. PACKAGING AN APPLICATION

Abstract

Applications must be packed as an OSGi bundle before they can be deployed into Red Hat JBoss Fuse. You will not need to include any Apache CXF specific packages in your bundle. The Apache CXF packages are included in JBoss Fuse. You need to ensure you import the required packages when building your bundle.

CREATING A BUNDLE

To deploy a Apache CXF application into Red Hat JBoss Fuse, you need to package it as an OSGi bundle. There are several tools available for assisting in the process. JBoss Fuse uses the Maven bundle plug-in whose use is described in Appendix D, *Using the Maven OSGi Tooling*.

REQUIRED BUNDLE

The Apache CXF runtime components are included in JBoss Fuse as an OSGi bundle called **`org.apache.cxf.cxf-bundle`**. This bundle needs to be installed in the JBoss Fuse container before your application's bundle can be started.

To inform the container of this dependency, you use the OSGi manifest's Required-Bundle property.

REQUIRED PACKAGES

In order for your application to use the Apache CXF components, you need to import their packages into the application's bundle. Because of the complex nature of the dependencies in Apache CXF, you cannot rely on the Maven bundle plug-in, or the **bnd** tool, to automatically determine the needed imports. You will need to explicitly declare them.

You need to import the following packages into your bundle:

- javax.jws
- javax.wsdl
- META-INF.cxf
- META-INF.cxf.osgi
- org.apache.cxf.bus
- org.apache.cxf.bus.spring
- org.apache.cxf.bus.resource
- org.apache.cxf.configuration.spring
- org.apache.cxf.resource
- org.apache.servicemix.cxf.transport.http_osgi
- org.springframework.beans.factory.config

EXAMPLE

Example 21.1, “Apache CXF Application Manifest” shows a manifest for a Apache CXF application's OSGi bundle.

Example 21.1. Apache CXF Application Manifest

```
Manifest-Version: 1.0
Built-By: FinnMcCumial
Created-By: Apache Maven Bundle Plugin
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0.txt
Import-Package: javax.jws,javax.wsdl,META-INF.cxf,META-INF.cxf.osgi,
org.apache.cxf.bus,org.apache.cxf.bus.spring,org.apache.bus.resource,
org.apache.cxf.configuration.spring, org.apache.cxf.resource,
org.apache.servicemix.cxf.transport.http_cxf,
org.springframework.beans.factory.config
Bnd-LastModified: 1222079507224
Bundle-Version: 4.0.0.fuse
Bundle-Name: Fuse CXF Example
Bundle-Description: This is a sample CXF manifest.
Build-Jdk: 1.5.0_08
Private-Package: org.apache.servicemix.examples.cxf
Required-Bundle: org.apache.cxf.cxf-bundle
Bundle-ManifestVersion: 2
Bundle-SymbolicName: cxf-wsdl-first-osgi
Tool: Bnd-0.0.255
```

CHAPTER 22. DEPLOYING AN APPLICATION

Abstract

Red Hat JBoss Fuse will automatically install and deploy your application. You can also manually control the state of your application using the console.

OVERVIEW

There are two ways to deploy your application into Red Hat JBoss Fuse:

1. Rely on the hot deployment mechanism.
2. Use the console.

You can also start and stop a deployed application using the console.

HOT DEPLOYMENT

The easiest way to deploy an application is to place it in the hot deployment folder. By default, the hot deployment folder is *InstallDir*/**deploy**. Any bundle placed in this folder is installed into the container. If its dependencies can be resolved, the bundle is activated.

One the bundle is installed in the container, you can manage it using the console.

DEPLOYING FROM THE CONSOLE

The easiest way to deploy an application from the console is to install it and start it in one step. This is done using the **osgi install -s** command. It takes the location of the bundle as a URI. So the command:

```
servicemix>osgi install -s file:/home/finn/ws/widgetapp.jar
```

Installs and attempts to start the bundle **widgetapp.jar** which is located in **/home/finn/ws**.

You can use the **osgi install** command without the **-s** flag. That will install the bundle without attempting to start it. You will then have to manually start the bundle using the **osgi start** command.

The **osgi start** command uses the bundle ID to determine which bundle to activate. [1]

REFRESHING AN APPLICATION

If you make changes to your application and want to redeploy it, you can do so by replacing the installed bundle with a new version and using the **osgi refresh** command. This command instructs the container to stop the running instance of your application, reload the bundle, and restart it.

The **osgi refresh** command uses a bundle ID to determine which bundle to refresh. [1]

STOPPING AN APPLICATION

If you want to temporarily deactivate your application you can use the **`osgi stop`** command. The **`osgi stop`** moves your application's bundle from the active state to the resolved state. This means that it can be easily restarted using the **`osgi start`** command.

The **`osgi stop`** command uses a bundle ID to determine which bundle to stop. [1]

UNINSTALLING AN APPLICATION

When you want to permanently remove an application from the container you need to uninstall it. Bundles can only be installed when they are not active. This means that you have to stop your application using the **`osgi stop`** command before trying to unistall it.

Once the application's bundle is stopped, you can use the **`osgi uninstall`** command to remove the bundle from the container. This does not delete the physical bundle. It just removes the bundle from the container's list of installed bundles.

The **`osgi stop`** command uses a bundle ID to determine which bundle to unistall. [1]

---

[1] You can get a list of the bundle IDs using the **`osgi list`** command.

APPENDIX C. APACHE CXF BINDING IDS

Table C.1. Binding IDs for Message Bindings

Binding	ID
CORBA	`http://cxf.apache.org/bindings/corba`
HTTP/REST	`http://apache.org/cxf/binding/http`
SOAP 1.1	`http://schemas.xmlsoap.org/wsdl/soap/http`
SOAP 1.1 w/ MTOM	`http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true`
SOAP 1.2	`http://www.w3.org/2003/05/soap/bindings/HTTP/`
SOAP 1.2 w/ MTOM	`http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true`
XML	`http://cxf.apache.org/bindings/xformat`

APPENDIX D. USING THE MAVEN OSGI TOOLING

Abstract

Manually creating a bundle, or a collection of bundles, for a large project can be cumbersome. The Maven bundle plug-in makes the job easier by automating the process and providing a number of shortcuts for specifying the contents of the bundle manifest.

The Red Hat JBoss Fuse OSGi tooling uses the Maven bundle plug-in from Apache Felix. The bundle plug-in is based on the **bnd** tool from Peter Kriens. It automates the construction of OSGi bundle manifests by introspecting the contents of the classes being packaged in the bundle. Using the knowledge of the classes contained in the bundle, the plug-in can calculate the proper values to populate the Import-Packages and the Export-Package properties in the bundle manifest. The plug-in also has default values that are used for other required properties in the bundle manifest.

To use the bundle plug-in, do the following:

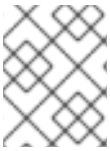
1. Add the bundle plug-in to your project's POM file.
2. Configure the plug-in to correctly populate your bundle's manifest.

D.1. SETTING UP A RED HAT JBOSS FUSE OSGI PROJECT

Overview

A Maven project for building an OSGi bundle can be a simple single level project. It does not require any sub-projects. However, it does require that you do the following:

1. Add the bundle plug-in to your POM.
2. Instruct Maven to package the results as an OSGi bundle.



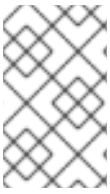
NOTE

There are several Maven archetypes you can use to set up your project with the appropriate settings.

Directory structure

A project that constructs an OSGi bundle can be a single level project. It only requires that you have a top-level POM file and a `src` folder. As in all Maven projects, you place all Java source code in the `src/java` folder, and you place any non-Java resources in the `src/resources` folder.

Non-Java resources include Spring configuration files, JBI endpoint configuration files, and WSDL contracts.



NOTE

Red Hat JBoss Fuse OSGi projects that use Apache CXF, Apache Camel, or another Spring configured bean also include a `beans.xml` file located in the `src/resources/META-INF/spring` folder.

Adding a bundle plug-in

Before you can use the bundle plug-in you must add a dependency on Apache Felix. After you add the dependency, you can add the bundle plug-in to the plug-in portion of the POM.

Example D.1, "Adding an OSGi bundle plug-in to a POM" shows the POM entries required to add the bundle plug-in to your project.

Example D.1. Adding an OSGi bundle plug-in to a POM

```
...
<dependencies>
1   <dependency>
      <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
2    <plugin>
       <groupId>org.apache.felix</groupId>
     <artifactId>maven-bundle-plugin</artifactId>
     <configuration>
       <instructions>
         <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
3          <Import-Package>*,org.apache.camel.osgi</Import-Package>
4            <Private-
5 Package>org.apache.servicemix.examples.camel</Private-Package>
             </instructions>
     </configuration>
    </plugin>
  </plugins>
</build>
...
```

The entries in Example D.1, "Adding an OSGi bundle plug-in to a POM" do the following:

1. Adds the dependency on Apache Felix
2. Adds the bundle plug-in to your project
3. Configures the plug-in to use the project's artifact ID as the bundle's symbolic name
4. Configures the plug-in to include all Java packages imported by the bundled classes; also imports the org.apache.camel.osgi package
5. Configures the plug-in to bundle the listed class, but not to include them in the list of exported packages

NOTE

Edit the configuration to meet the requirements of your project.

For more information on configuring the bundle plug-in, see Section D.2, “Configuring the Bundle Plug-In”.

Activating a bundle plug-in

To have Maven use the bundle plug-in, instruct it to package the results of the project as a bundle. Do this by setting the POM file's **packaging** element to **bundle**.

Useful Maven archetypes

There are several Maven archetypes to generate a project that is preconfigured to use the bundle plug-in:

- the section called “Spring OSGi archetype”
- the section called “Apache CXF code-first archetype”
- the section called “Apache CXF wsdl-first archetype”
- the section called “Apache Camel archetype”

Spring OSGi archetype

The Spring OSGi archetype creates a generic project for building an OSGi project using Spring DM, as shown:

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

You invoke the archetype using the following command:

```
mvn archetype:create -DarchetypeGroupId=org.springframework.osgi -
DarchetypeArtifactId=spring-osgi-bundle-archetype -DarchetypeVersion=1.12
-DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

Apache CXF code-first archetype

The Apache CXF code-first archetype creates a project for building a service from Java, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-
archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=spring-osgi-bundle-archetype -
DarchetypeVersion=2008.01.0.3-fuse -DgroupId=groupId -
DartifactId=artifactId -Dversion=version
```

Apache CXF wsdl-first archetype

The Apache CXF wsdl-first archetype creates a project for creating a service from WSDL, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-
archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype -
DarchetypeVersion=2008.01.0.3-fuse -DgroupId=groupId -
DartifactId=artifactId -Dversion=version
```

Apache Camel archetype

The Apache Camel archetype creates a project for building a route that is deployed into JBoss Fuse, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2008.01.0.3-
fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-camel-archetype -
DarchetypeVersion=2008.01.0.3-fuse -DgroupId=groupId -
DartifactId=artifactId -Dversion=version
```

D.2. CONFIGURING THE BUNDLE PLUG-IN

Overview

A bundle plug-in requires very little information to function. All of the required properties use default settings to generate a valid OSGi bundle.

While you can create a valid bundle using just the default values, you will probably want to modify some of the values. You can specify most of the properties inside the plug-in's **`instructions`** element.

Configuration properties

Some of the commonly used configuration properties are:

- Bundle-SymbolicName
- Bundle-Name
- Bundle-Version
- Export-Package
- Private-Package

- Import-Package

Setting a bundle's symbolic name

By default, the bundle plug-in sets the value for the Bundle-SymbolicName property to *groupId* `+ "."` `+` *artifactId*, with the following exceptions:

- If *groupId* has only one section (no dots), the first package name with classes is returned.

  For example, if the group Id is **`commons-logging:commons-logging`**, the bundle's symbolic name is **`org.apache.commons.logging`**.

- If *artifactId* is equal to the last section of *groupId*, then *groupId* is used.

  For example, if the POM specifies the group ID and artifact ID as **`org.apache.maven:maven`**, the bundle's symbolic name is **`org.apache.maven`**.

- If *artifactId* starts with the last section of *groupId*, that portion is removed.

  For example, if the POM specifies the group ID and artifact ID as **`org.apache.maven:maven-core`**, the bundle's symbolic name is **`org.apache.maven.core`**.

To specify your own value for the bundle's symbolic name, add a **`Bundle-SymbolicName`** child in the plug-in's **`instructions`** element, as shown in Example D.2.

Example D.2. Setting a bundle's symbolic name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's name

By default, a bundle's name is set to **`${project.name}`**.

To specify your own value for the bundle's name, add a **`Bundle-Name`** child to the plug-in's **`instructions`** element, as shown in Example D.3.

Example D.3. Setting a bundle's name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
```

```
      <Bundle-Name>JoeFred</Bundle-Name>
      ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's version

By default, a bundle's version is set to **`${project.version}`**. Any dashes (**`-`**) are replaced with dots (**`.`**) and the number is padded up to four digits. For example, **`4.2-SNAPSHOT`** becomes **`4.2.0.SNAPSHOT`**.

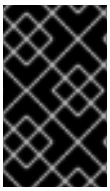
To specify your own value for the bundle's version, add a **`Bundle-Version`** child to the plug-in's **`instructions`** element, as shown in Example D.4.

Example D.4. Setting a bundle's version

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying exported packages

By default, the OSGi manifest's **`Export-Package`** list is populated by all of the packages in your local Java source code (under **`src/main/java`**), *except* for the deault package, **`.`**, and any packages containing **`.impl`** or **`.internal`**.



IMPORTANT

If you use a **`Private-Package`** element in your plug-in configuration and you do not specify a list of packages to export, the default behavior includes only the packages listed in the **`Private-Package`** element in the bundle. No packages are exported.

The default behavior can result in very large packages and in exporting packages that should be kept private. To change the list of exported packages you can add an **`Export-Package`** child to the plug-in's **`instructions`** element.

The **`Export-Package`** element specifies a list of packages that are to be included in the bundle and that are to be exported. The package names can be specified using the **`*`** wildcard symbol. For example, the entry **`com.fuse.demo.*`** includes all packages on the project's classpath that start with com.fuse.demo.

You can specify packages to be excluded be prefixing the entry with `!`. For example, the entry `!com.fuse.demo.private` excludes the package com.fuse.demo.private.

When excluding packages, the order of entries in the list is important. The list is processed in order from the beginning and any subsequent contradicting entries are ignored.

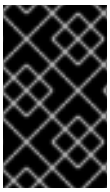
For example, to include all packages starting with com.fuse.demo except the package com.fuse.demo.private, list the packages using:

```
!com.fuse.demo.private,com.fuse.demo.*
```

However, if you list the packages using com.fuse.demo.*,!com.fuse.demo.private, then com.fuse.demo.private is included in the bundle because it matches the first pattern.

Specifying private packages

If you want to specify a list of packages to include in a bundle *without* exporting them, you can add a **`Private-Package`** instruction to the bundle plug-in configuration. By default, if you do not specify a **`Private-Package`** instruction, all packages in your local Java source are included in the bundle.



IMPORTANT

If a package matches an entry in both the **`Private-Package`** element and the **`Export-Package`** element, the **`Export-Package`** element takes precedence. The package is added to the bundle and exported.

The **`Private-Package`** element works similarly to the **`Export-Package`** element in that you specify a list of packages to be included in the bundle. The bundle plug-in uses the list to find all classes on the project's classpath that are to be included in the bundle. These packages are packaged in the bundle, but not exported (unless they are also selected by the **`Export-Package`** instruction).

Example D.5 shows the configuration for including a private package in a bundle

Example D.5. Including a private package in a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying imported packages

By default, the bundle plug-in populates the OSGi manifest's **`Import-Package`** property with a list of all the packages referred to by the contents of the bundle.

While the default behavior is typically sufficient for most projects, you might find instances where you want to import packages that are not automatically added to the list. The default behavior can also result in unwanted packages being imported.

To specify a list of packages to be imported by the bundle, add an **Import-Package** child to the plug-in's **instructions** element. The syntax for the package list is the same as for the **Export-Package** element and the **Private-Package** element.

IMPORTANT

When you use the **Import-Package** element, the plug-in does not automatically scan the bundle's contents to determine if there are any required imports. To ensure that the contents of the bundle are scanned, you must place an * as the last entry in the package list.

Example D.6 shows the configuration for specifying the packages imported by a bundle

Example D.6. Specifying the packages imported by a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Import-Package>javax.jws,
         javax.wsdl,
         org.apache.cxf.bus,
         org.apache.cxf.bus.spring,
         org.apache.cxf.bus.resource,
         org.apache.cxf.configuration.spring,
         org.apache.cxf.resource,
         org.springframework.beans.factory.config,
         *
     </Import-Package>
     ...
   </instructions>
  </configuration>
</plugin>
```

More information

For more information on configuring a bundle plug-in, see:

- "Managing OSGi Dependencies"
- Apache Felix documentation
- Peter Kriens' aQute Software Consultancy web site

PART V. DEVELOPING APPLICATIONS USING JAX-WS

Abstract

This guide describes how to develop Web services using the standard JAX-WS APIs.

CHAPTER 23. BOTTOM-UP SERVICE DEVELOPMENT

Abstract

There are many instances where you have Java code that already implements a set of functionality that you want to expose as part of a service oriented application. You may also simply want to avoid using WSDL to define your interface. Using JAX-WS annotations, you can add the information required to service enable a Java class. You can also create a *Service Endpoint Interface* (SEI) that can be used in place of a WSDL contract. If you want a WSDL contract, Apache CXF provides tools to generate a contract from annotated Java code.

To create a service starting from Java you must do the following:

1. Create a Service Endpoint Interface (SEI) that defines the methods you want to expose as a service.

   TIP

   You can work directly from a Java class, but working from an interface is the recommended approach. Interfaces are better suited for sharing with the developers who are responsible for developing the applications consuming your service. The interface is smaller and does not provide any of the service's implementation details.

2. Add the required annotations to your code.
3. Generate the WSDL contract for your service.

   TIP

   If you intend to use the SEI as the service's contract, it is not necessary to generate a WSDL contract.

4. Publish the service as a service provider.

23.1. CREATING THE SEI

Overview

The *service endpoint interface* (SEI) is the piece of Java code that is shared between a service implementation and the consumers that make requests on that service. The SEI defines the methods implemented by the service and provides details about how the service will be exposed as an endpoint. When starting with a WSDL contract, the SEI is generated by the code generators. However, when starting from Java, it is the developer's responsibility to create the SEI.

There are two basic patterns for creating an SEI:

- Green field development — In this pattern, you are developing a new service without any existing Java code or WSDL. It is best to start by creating the SEI. You can then distribute the SEI to any developers that are responsible for implementing the service providers and consumers that use the SEI.

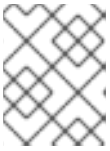
NOTE

The recommended way to do green field service development is to start by creating a WSDL contract that defines the service and its interfaces. See Chapter 25, *A Starting Point WSDL Contract*

- Service enablement — In this pattern, you typically have an existing set of functionality that is implemented as a Java class, and you want to service enable it. This means that you must do two things:

  1. Create an SEI that contains **only** the operations that are going to be exposed as part of the service.

  2. Modify the existing Java class so that it implements the SEI.



NOTE

Although you can add the JAX-WS annotations to a Java class, it is not recommended.

Writing the interface

The SEI is a standard Java interface. It defines a set of methods that a class implements. It can also define a number of member fields and constants to which the implementing class has access.

In the case of an SEI the methods defined are intended to be mapped to operations exposed by a service. The SEI corresponds to a **`wsdl:portType`** element. The methods defined by the SEI correspond to **`wsdl:operation`** elements in the **`wsdl:portType`** element.

TIP

JAX-WS defines an annotation that allows you to specify methods that are not exposed as part of a service. However, the best practice is to leave those methods out of the SEI.

Example 23.1, “Simple SEI” shows a simple SEI for a stock updating service.

Example 23.1. Simple SEI

```
package com.fusesource.demo;

public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

Implementing the interface

Because the SEI is a standard Java interface, the class that implements it is a standard Java class. If you start with a Java class you must modify it to implement the interface. If you start with the SEI, the implementation class implements the SEI.

Example 23.2, “Simple Implementation Class” shows a class for implementing the interface in Example 23.1, “Simple SEI”.

Example 23.2. Simple Implementation Class

```
package com.fusesource.demo;

import java.util.*;

public class stockQuoteReporter implements quoteReporter
{
  ...
public Quote getQuote(String ticker)
  {
    Quote retVal = new Quote();
    retVal.setID(ticker);
    retVal.setVal(Board.check(ticker));[2]
    Date retDate = new Date();
    retVal.setTime(retDate.toString());
    return(retVal);
  }
}
```

23.2. ANNOTATING THE CODE

The JAX-WS annotations specify the metadata used to map the SEI to a fully specified service definition. Among the information provided in the annotations are the following:

- The target namespace for the service.
- The name of the class used to hold the request message
- The name of the class used to hold the response message
- If an operation is a one way operation
- The binding style the service uses
- The name of the class used for any custom exceptions
- The namespaces under which the types used by the service are defined

TIP

Most of the annotations have sensible defaults and it is not necessary to provide values for them. However, the more information you provide in the annotations, the better your service definition is specified. A well-specified service definition increases the likelihood that all parts of a distributed application will work together.

23.2.1. Required Annotations

Overview

In order to create a service from Java code you are only required to add one annotation to your code. You must add the **@WebService** annotation on both the SEI and the implementation class.

The @WebService annotation

The **@WebService** annotation is defined by the **javax.jws.WebService** interface and it is placed on an interface or a class that is intended to be used as a service. **@WebService** has the properties described in Table 23.1, "**@WebService** Properties"

Table 23.1. @WebService Properties

Property	Description
name	Specifies the name of the service interface. This property is mapped to the **name** attribute of the **wsdl:portType** element that defines the service's interface in a WSDL contract. The default is to append **PortType** to the name of the implementation class. [a]
targetNamespace	Specifies the target namespace where the service is defined. If this property is not specified, the target namespace is derived from the package name.
serviceName	Specifies the name of the published service. This property is mapped to the **name** attribute of the **wsdl:service** element that defines the published service. The default is to use the name of the service's implementation class. [a]
wsdlLocation	Specifies the URL where the service's WSDL contract is stored. This must be specified using a relative URL. The default is the URL where the service is deployed.
endpointInterface	Specifies the full name of the SEI that the implementation class implements. This property is only specified when the attribute is used on a service implementation class.
portName	Specifies the name of the endpoint at which the service is published. This property is mapped to the **name** attribute of the **wsdl:port** element that specifies the endpoint details for a published service. The default is the append **Port** to the name of the service's implementation class.[a]
[a] When you generate WSDL from an SEI the interface's name is used in place of the implementation class' name.	

TIP

It is not necessary to provide values for any of the **@WebService** annotation's properties. However, it is recommended that you provide as much information as you can.

Annotating the SEI

The SEI requires that you add the **@WebService** annotation. Because the SEI is the contract that defines the service, you should specify as much detail as possible about the service in the **@WebService** annotation's properties.

Example 23.3, "Interface with the **@WebService** Annotation" shows the interface defined in Example 23.1, "Simple SEI" with the **@WebService** annotation.

Example 23.3. Interface with the @WebService Annotation

```
package com.fusesource.demo;

import javax.jws.*;

1 @WebService(name="quoteUpdater",
2              targetNamespace="http:\\demos.redhat.com",
3             serviceName="updateQuoteService",

4 wsdlLocation="http:\\demos.redhat.com\quoteExampleService?wsdl",
5              portName="updateQuotePort")
    public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

The **@WebService** annotation in Example 23.3, "Interface with the **@WebService** Annotation" does the following:

1. Specifies that the value of the **name** attribute of the **wsdl:portType** element defining the service interface is **quoteUpdater**.
2. Specifies that the target namespace of the service is http:\\demos.redhat.com.
3. Specifies that the value of the **name** of the **wsdl:service** element defining the published service is **updateQuoteService**.
4. Specifies that the service will publish its WSDL contract at http:\\demos.redhat.com\quoteExampleService?wsdl.
5. Specifies that the value of the **name** attribute of the **wsdl:port** element defining the endpoint exposing the service is **updateQuotePort**.

Annotating the service implementation

In addition to annotating the SEI with the **@WebService** annotation, you also must annotate the service implementation class with the **@WebService** annotation. When adding the annotation to the

service implementation class you only need to specify the endpointInterface property. As shown in Example 23.4, "Annotated Service Implementation Class" the property must be set to the full name of the SEI.

Example 23.4. Annotated Service Implementation Class

```
package org.eric.demo;

import javax.jws.*;

@WebService(endpointInterface="com.fusesource.demo.quoteReporter")
public class stockQuoteReporter implements quoteReporter
{
public Quote getQuote(String ticker)
  {
  ...
  }
}
```

23.2.2. Optional Annotations

While the **@WebService** annotation is sufficient for service enabling a Java interface or a Java class, it does not fully describe how the service will be exposed as a service provider. The JAX-WS programming model uses a number of optional annotations for adding details about your service, such as the binding it uses, to the Java code. You add these annotations to the service's SEI.

TIP

The more details you provide in the SEI the easier it is for developers to implement applications that can use the functionality it defines. It also makes the WSDL documents generated by the tools more specific.

23.2.2.1. Defining the Binding Properties with Annotations

Overview

If you are using a SOAP binding for your service, you can use JAX-WS annotations to specify a number of the bindings properties. These properties correspond directly to the properties you can specify in a service's WSDL contract. Some of the settings, such as the parameter style, can restrict how you implement a method. These settings can also effect which annotations can be used when annotating method parameters.

The @SOAPBinding annotation

The **@SOAPBinding** annotation is defined by the **javax.jws.soap.SOAPBinding** interface. It provides details about the SOAP binding used by the service when it is deployed. If the **@SOAPBinding** annotation is not specified, a service is published using a wrapped doc/literal SOAP binding.

You can put the **@SOAPBinding** annotation on the SEI and any of the SEI's methods. When it is used on a method, setting of the method's **@SOAPBinding** annotation take precedence.

Table 23.2, "**@SOAPBinding** Properties" shows the properties for the **@SOAPBinding** annotation.

Table 23.2. @SOAPBinding Properties

Property	Values	Description
style	**Style.DOCUMENT** (default) **Style.RPC**	Specifies the style of the SOAP message. If **RPC** style is specified, each message part within the SOAP body is a parameter or return value and appears inside a wrapper element within the **soap:body** element. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. If **DOCUMENT** style is specified, the contents of the SOAP body must be a valid XML document, but its form is not as tightly constrained.
use	**Use.LITERAL** (default) **Use.ENCODED**[a]	Specifies how the data of the SOAP message is streamed.
parameterStyle [b]	**ParameterStyle.BARE** **ParameterStyle.WRAPPED** (default)	Specifies how the method parameters, which correspond to message parts in a WSDL contract, are placed into the SOAP message body. If **BARE** is specified, each parameter is placed into the message body as a child element of the message root. If **WRAPPED** is specified, all of the input parameters are wrapped into a single element on a request message and all of the output parameters are wrapped into a single element in the response message.

[a] **Use.ENCODED** is not currently supported.

[b] If you set the style to **RPC** you must use the **WRAPPED** parameter style.

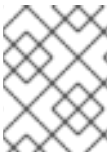
Document bare style parameters

Document bare style is the most direct mapping between Java code and the resulting XML representation of the service. When using this style, the schema types are generated directly from the input and output parameters defined in the operation's parameter list.

You specify you want to use bare document\literal style by using the **@SOAPBinding** annotation with its style property set to **Style.DOCUMENT**, and its parameterStyle property set to **ParameterStyle.BARE**.

To ensure that an operation does not violate the restrictions of using document style when using bare parameters, your operations must adhere to the following conditions:

- The operation must have no more than one input or input/output parameter.
- If the operation has a return type other than void, it must not have any output or input/output parameters.
- If the operation has a return type of void, it must have no more than one output or input/output parameter.



NOTE

Any parameters that are placed in the SOAP header using the **@WebParam** annotation or the **@WebResult** annotation are not counted against the number of allowed parameters.

Document wrapped parameters

Document wrapped style allows a more RPC like mapping between the Java code and the resulting XML representation of the service. When using this style, the parameters in the method's parameter list are wrapped into a single element by the binding. The disadvantage of this is that it introduces an extra-layer of indirection between the Java implementation and how the messages are placed on the wire.

To specify that you want to use wrapped document\literal style use the **@SOAPBinding** annotation with its style property set to **Style.DOCUMENT**, and its parameterStyle property set to **ParameterStyle.WRAPPED**.

You have some control over how the wrappers are generated by using the the section called “The @RequestWrapper annotation” annotation and the the section called “The @ResponseWrapper annotation” annotation.

Example

Example 23.5, “Specifying a Document Bare SOAP Binding with the SOAP Binding Annotation” shows an SEI that uses document bare SOAP messages.

Example 23.5. Specifying a Document Bare SOAP Binding with the SOAP Binding Annotation

```
package org.eric.demo;

import javax.jws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;

@WebService(name="quoteReporter")
@SOAPBinding(parameterStyle=ParameterStyle.BARE)
public interface quoteReporter
{
  ...
}
```

23.2.2.2. Defining Operation Properties with Annotations

Overview

When the runtime maps your Java method definitions into XML operation definitions it provides details such as:

- What the exchanged messages look like in XML
- If the message can be optimized as a one way message
- The namespaces where the messages are defined

The `@WebMethod` annotation

The **`@WebMethod`** annotation is defined by the **`javax.jws.WebMethod`** interface. It is placed on the methods in the SEI. The **`@WebMethod`** annotation provides the information that is normally represented in the **`wsdl:operation`** element describing the operation to which the method is associated.

Table 23.3, "**`@WebMethod`** Properties" describes the properties of the **`@WebMethod`** annotation.

Table 23.3. `@WebMethod` Properties

Property	Description
operationName	Specifies the value of the associated **`wsdl:operation`** element's **`name`**. The default value is the name of the method.
action	Specifies the value of the **`soapAction`** attribute of the **`soap:operation`** element generated for the method. The default value is an empty string.
exclude	Specifies if the method should be excluded from the service interface. The default is **`false`**.

The `@RequestWrapper` annotation

The **`@RequestWrapper`** annotation is defined by the **`javax.xml.ws.RequestWrapper`** interface. It is placed on the methods in the SEI. The **`@RequestWrapper`** annotation specifies the Java class implementing the wrapper bean for the method parameters of the request message starting a message exchange. It also specifies the element names, and namespaces, used by the runtime when marshalling and unmarshalling the request messages.

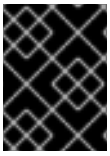
Table 23.4, "**`@RequestWrapper`** Properties" describes the properties of the **`@RequestWrapper`** annotation.

Table 23.4. `@RequestWrapper` Properties

Property	Description
localName	Specifies the local name of the wrapper element in the XML representation of the request message. The default value is either the name of the method, or the value of the the section called "The `@WebMethod` annotation" annotation's operationName property.
targetNamespace	Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.
className	Specifies the full name of the Java class that implements the wrapper element.

TIP

Only the className property is required.



IMPORTANT

If the method is also annotated with the **`@SOAPBinding`** annotation, and its parameterStyle property is set to **`ParameterStyle.BARE`**, this annotation is ignored.

The `@ResponseWrapper` annotation

The **`@ResponseWrapper`** annotation is defined by the **`javax.xml.ws.ResponseWrapper`** interface. It is placed on the methods in the SEI. The **`@ResponseWrapper`** specifies the Java class implementing the wrapper bean for the method parameters in the response message in the message exchange. It also specifies the element names, and namespaces, used by the runtime when marshaling and unmarshalling the response messages.

Table 23.5, "**`@ResponseWrapper`** Properties" describes the properties of the **`@ResponseWrapper`** annotation.

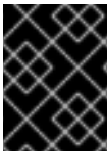
Table 23.5. `@ResponseWrapper` Properties

Property	Description
localName	Specifies the local name of the wrapper element in the XML representation of the response message. The default value is either the name of the method with **Response** appended, or the value of the the section called "The `@WebMethod` annotation" annotation's operationName property with **Response** appended.
targetNamespace	Specifies the namespace where the XML wrapper element is defined. The default value is the target namespace of the SEI.

Property	Description
className	Specifies the full name of the Java class that implements the wrapper element.

TIP

Only the className property is required.



IMPORTANT

If the method is also annotated with the **@SOAPBinding** annotation and its parameterStyle property is set to **ParameterStyle.BARE**, this annotation is ignored.

The @WebFault annotation

The **@WebFault** annotation is defined by the **javax.xml.ws.WebFault** interface. It is placed on exceptions that are thrown by your SEI. The **@WebFault** annotation is used to map the Java exception to a **wsdl:fault** element. This information is used to marshall the exceptions into a representation that can be processed by both the service and its consumers.

Table 23.6, "**@WebFault** Properties" describes the properties of the **@WebFault** annotation.

Table 23.6. @WebFault Properties

Property	Description
name	Specifies the local name of the fault element.
targetNamespace	Specifies the namespace under which the fault element is defined. The default value is the target namespace of the SEI.
faultName	Specifies the full name of the Java class that implements the exception.

IMPORTANT

The name property is required.

The @Oneway annotation

The **@Oneway** annotation is defined by the **javax.jws.Oneway** interface. It is placed on the methods in the SEI that will not require a response from the service. The **@Oneway** annotation tells the run time that it can optimize the execution of the method by not waiting for a response and by not reserving any resources to process a response.

This annotation can only be used on methods that meet the following criteria:

- They return void

- They have no parameters that implement the **`Holder`** interface
- They do not throw any exceptions that can be passed back to a consumer

Example

Example 23.6, "SEI with Annotated Methods" shows an SEI with its methods annotated.

Example 23.6. SEI with Annotated Methods

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;

@WebService(name="quoteReporter")
public interface quoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                   className="org.eric.demo.Quote")
  public Quote getQuote(String ticker);
}
```

23.2.2.3. Defining Parameter Properties with Annotations

Overview

The method parameters in the SEI correspond to the **`wsdl:message`** elements and their **`wsdl:part`** elements. JAX-WS provides annotations that allow you to describe the **`wsdl:part`** elements that are generated for the method parameters.

The `@WebParam` annotation

The **`@WebParam`** annotation is defined by the **`javax.jws.WebParam`** interface. It is placed on the parameters of the methods defined in the SEI. The **`@WebParam`** annotation allows you to specify the direction of the parameter, if the parameter will be placed in the SOAP header, and other properties of the generated **`wsdl:part`**.

Table 23.7, "**`@WebParam`** Properties" describes the properties of the **`@WebParam`** annotation.

Table 23.7. `@WebParam` Properties

Property	Values	Description

Property	Values	Description
name		Specifies the name of the parameter as it appears in the generated WSDL document. For RPC bindings, this is the name of the **wsdl:part** representing the parameter. For document bindings, this is the local name of the XML element representing the parameter. Per the JAX-WS specification, the default is **arg*N***, where *N* is replaced with the zero-based argument index (i.e., arg0, arg1, etc.).
targetNamespace		Specifies the namespace for the parameter. It is only used with document bindings where the parameter maps to an XML element. The default is to use the service's namespace.
mode	**Mode.IN** (default)[a] **Mode.OUT** **Mode.INOUT**	Specifies the direction of the parameter.
header	**false** (default) **true**	Specifies if the parameter is passed as part of the SOAP header.
partName		Specifies the value of the **name** attribute of the **wsdl:part** element for the parameter. This property is used for document style SOAP bindings.

[a] Any parameter that implements the **Holder** interface is mapped to **Mode.INOUT** by default.

The @WebResult annotation

The **@WebResult** annotation is defined by the **javax.jws.WebResult** interface. It is placed on the methods defined in the SEI. The **@WebResult** annotation allows you to specify the properties of the **wsdl:part** that is generated for the method's return value.

Table 23.8, "**@WebResult** Properties" describes the properties of the **@WebResult** annotation.

Table 23.8. @WebResult Properties

Property	Description
name	Specifies the name of the return value as it appears in the generated WSDL document. For RPC bindings, this is the name of the `wsdl:part` representing the return value. For document bindings, this is the local name of the XML element representing the return value. The default value is `return`.
targetNamespace	Specifies the namespace for the return value. It is only used with document bindings where the return value maps to an XML element. The default is to use the service's namespace.
header	Specifies if the return value is passed as part of the SOAP header.
partName	Specifies the value of the `name` attribute of the `wsdl:part` element for the return value. This property is used for document style SOAP bindings.

Example

Example 23.7, "Fully Annotated SEI" shows an SEI that is fully annotated.

Example 23.7. Fully Annotated SEI

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;
import javax.jws.WebParam.*;

@WebService(targetNamespace="http://demo.redhat.com",
            name="quoteReporter")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface quoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                   className="org.eric.demo.Quote")
  @WebResult(targetNamespace="http://demo.redhat.com/types",
             name="updatedQuote")
  public Quote getQuote(

@WebParam(targetNamespace="http://demo.redhat.com/types",
                                  name="stockTicker",
                                  mode=Mode.IN)
```

```
                        String ticker
    );
}
```

23.2.3. Apache CXF Annotations

23.2.3.1. WSDL Documentation

@WSDLDocumentation annotation

The **@WSDLDocumentation** annotation is defined by the **org.apache.cxf.annotations.WSDLDocumentation** interface. It can be placed on the SEI or the SEI methods.

This annotation enables you to add documentation, which will then appear within **wsdl:documentation** elements after the SEI is converted to WSDL. By default, the documentation elements appear inside the port type, but you can specify the placement property to make the documentation appear at other locations in the WSDL file. Table 23.9, "@WSDLDocumentation properties" shows the properties supported by the **@WSDLDocumentation** annotation.

Table 23.9. @WSDLDocumentation properties

Property	Description
value	*(Required)* A string containing the documentation text.
placement	*(Optional)* Specifies where in the WSDL file this documentation is to appear. For the list of possible placement values, see the section called "Placement in the WSDL contract".
faultClass	*(Optional)* If the placement is set to be **FAULT_MESSAGE**, **PORT_TYPE_OPERATION_FAULT**, or **BINDING_OPERATION_FAULT**, you must also set this property to the Java class that represents the fault.

@WSDLDocumentationCollection annotation

The **@WSDLDocumentationCollection** annotation is defined by the **org.apache.cxf.annotations.WSDLDocumentationCollection** interface. It can be placed on the SEI or the SEI methods.

This annotation is used to insert multiple documentation elements at a single placement location or at various placement locations.

Placement in the WSDL contract

To specify where the documentation should appear in the WSDL contract, you can specify the **`placement`** property, which is of type **`WSDLDocumentation.Placement`**. The placement can have one of the following values:

- **`WSDLDocumentation.Placement.BINDING`**
- **`WSDLDocumentation.Placement.BINDING_OPERATION`**
- **`WSDLDocumentation.Placement.BINDING_OPERATION_FAULT`**
- **`WSDLDocumentation.Placement.BINDING_OPERATION_INPUT`**
- **`WSDLDocumentation.Placement.BINDING_OPERATION_OUTPUT`**
- **`WSDLDocumentation.Placement.DEFAULT`**
- **`WSDLDocumentation.Placement.FAULT_MESSAGE`**
- **`WSDLDocumentation.Placement.INPUT_MESSAGE`**
- **`WSDLDocumentation.Placement.OUTPUT_MESSAGE`**
- **`WSDLDocumentation.Placement.PORT_TYPE`**
- **`WSDLDocumentation.Placement.PORT_TYPE_OPERATION`**
- **`WSDLDocumentation.Placement.PORT_TYPE_OPERATION_FAULT`**
- **`WSDLDocumentation.Placement.PORT_TYPE_OPERATION_INPUT`**
- **`WSDLDocumentation.Placement.PORT_TYPE_OPERATION_OUTPUT`**
- **`WSDLDocumentation.Placement.SERVICE`**
- **`WSDLDocumentation.Placement.SERVICE_PORT`**
- **`WSDLDocumentation.Placement.TOP`**

Example of @WSDLDocumentation

Example 23.8, “Using @WSDLDocumentation” shows how to add a **`@WSDLDocumentation`** annotation to the SEI and to one of its methods.

Example 23.8. Using @WSDLDocumentation

```
@WebService
@WSDLDocumentation("A very simple example of an SEI")
public interface HelloWorld {
    @WSDLDocumentation("A traditional form of greeting")
    String sayHi(@WebParam(name = "text") String text);
}
```

When WSDL, shown in Example 23.9, "WSDL generated with documentation", is generated from the SEI in Example 23.8, "Using @WSDLDocumentation", the default placements of the **documentation** elements are, respectively, **PORT_TYPE** and **PORT_TYPE_OPERATION**.

Example 23.9. WSDL generated with documentation

```
<wsdl:definitions ... >
  ...
  <wsdl:portType name="HelloWorld">
 <wsdl:documentation>A very simple example of an
SEI</wsdl:documentation>
    <wsdl:operation name="sayHi">
 <wsdl:documentation>A traditional form of greeting</wsdl:documentation>
      <wsdl:input name="sayHi" message="tns:sayHi">
    </wsdl:input>
      <wsdl:output name="sayHiResponse" message="tns:sayHiResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

Example of @WSDLDocumentationCollection

Example 23.10, "Using @WSDLDocumentationCollection" shows how to add a **@WSDLDocumentationCollection** annotation to an SEI.

Example 23.10. Using @WSDLDocumentationCollection

```
@WebService
@WSDLDocumentationCollection(
    {
        @WSDLDocumentation("A very simple example of an SEI"),
        @WSDLDocumentation(value = "My top level documentation",
                           placement =
WSDLDocumentation.Placement.TOP),
        @WSDLDocumentation(value = "Binding documentation",
                           placement =
WSDLDocumentation.Placement.BINDING)
    }
)
public interface HelloWorld {
    @WSDLDocumentation("A traditional form of Geeky greeting")
    String sayHi(@WebParam(name = "text") String text);
}
```

23.2.3.2. Schema Validation of Messages

@SchemaValidation annotation

The **@SchemaValidation** annotation is defined by the **org.apache.cxf.annotations.SchemaValidation** interface. It is placed on the SEI.

This annotation turns on schema validation of the XML messages sent to this endpoint. This can be useful for testing purposes, when you suspect there is a problem with the format of incoming XML messages. By default, validation is disabled, because it has a significant impact on performance.

Example

Example 23.11, "Activating schema validation" shows how to enable schema validation of messages for endpoints based on the **HelloWorld** SEI.

Example 23.11. Activating schema validation

```
@WebService
@SchemaValidation
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

23.2.3.3. Specifying the Data Binding

@DataBinding annotation

The **@DataBinding** annotation is defined by the **org.apache.cxf.annotations.DataBinding** interface. It is placed on the SEI.

This annotation is used to associate a data binding with the SEI, replacing the default JAXB data binding. The value of the **@DataBinding** annotation must be the class that provides the data binding, ***ClassName*.class**.

Supported data bindings

The following data bindings are currently supported by Apache CXF:

- **org.apache.cxf.jaxb.JAXBDataBinding**

  (Default) The standard JAXB data binding.

- **org.apache.cxf.sdo.SDODataBinding**

  The Service Data Objects (SDO) data binding is based on the Apache Tuscany SDO implementation. If you want to use this data binding in the context of a Maven build, you need to add a dependency on the **cxf-rt-databinding-sdo** artifact.

- **org.apache.cxf.aegis.databinding.AegisDatabinding**

  If you want to use this data binding in the context of a Maven build, you need to add a dependency on the **cxf-rt-databinding-aegis** artifact.

- **org.apache.cxf.xmlbeans.XmlBeansDataBinding**

If you want to use this data binding in the context of a Maven build, you need to add a dependency on the **cxf-rt-databinding-xmlbeans** artifact.

- **org.apache.cxf.databinding.source.SourceDataBinding**

  This data binding belongs to the Apache CXF core.

- **org.apache.cxf.databinding.stax.StaxDataBinding**

  This data binding belongs to the Apache CXF core.

Example

Example 23.12, "Setting the data binding" shows how to associate the SDO binding with the **HelloWorld** SEI

Example 23.12. Setting the data binding

```
@WebService
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

23.2.3.4. Compressing Messages

@GZIP annotation

The **@GZIP** annotation is defined by the **org.apache.cxf.annotations.GZIP** interface. It is placed on the SEI.

Enables GZIP compression of messages. GZIP is a negotiated enhancement. That is, an initial request from a client will not be gzipped, but an **Accept** header will be added and, if the server supports GZIP compression, the response will be gzipped and any subsequent requests will be also.

Table 23.10, "@GZIP Properties" shows the optional properties supported by the **@GZIP** annotation.

Table 23.10. @GZIP Properties

Property	Description
threshold	Messages smaller than the size specified by this property are *not* gzipped. Default is -1 (no limit).

@FastInfoset

The **@FastInfoset** annotation is defined by the **org.apache.cxf.annotations.FastInfoset** interface. It is placed on the SEI.

Enables the use of FastInfoset format for messages. FastInfoset is a binary encoding format for XML, which aims to optimize both the message size and the processing performance of XML messages. For more details, see the following Sun article on Fast Infoset.

FastInfoset is a negotiated enhancement. That is, an initial request from a client will not be in FastInfoset format, but an **Accept** header will be added and, if the server supports FastInfoset, the response will be in FastInfoset and any subsequent requests will be also.

Table 23.11, "@FastInfoset Properties" shows the optional properties supported by the **@FastInfoset** annotation.

Table 23.11. @FastInfoset Properties

Property	Description
force	A boolean property that forces the use of FastInfoset format, instead of negotiating. When **true**, force the use of FastInfoset format; otherwise, negotiate. Default is **false**.

Example of @GZIP

Example 23.13, "Enabling GZIP" shows how to enable GZIP compression for the **HelloWorld** SEI.

Example 23.13. Enabling GZIP

```
@WebService
@GZIP
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

Exampe of @FastInfoset

Example 23.14, "Enabling FastInfoset" shows how to enable the FastInfoset format for the **HelloWorld** SEI.

Example 23.14. Enabling FastInfoset

```
@WebService
@FastInfoset
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

23.2.3.5. Enable Logging on an Endpoint

@Logging annotation

The **@Logging** annotation is defined by the **org.apache.cxf.annotations.Logging** interface. It is placed on the SEI.

This annotation enables logging for all endpoints associated with the SEI. Table 23.12, "@Logging Properties" shows the optional properties you can set in this annotation.

Table 23.12. @Logging Properties

Property	Description
limit	Specifies the size limit, beyond which the message is truncated in the logs. Default is 64K.
inLocation	Specifies the location to log incoming messages. Can be either **<stderr>**, **<stdout>**, **<logger>**, or a filename. Default is **<logger>**.
outLocation	Specifies the location to log outgoing messages. Can be either **<stderr>**, **<stdout>**, **<logger>**, or a filename. Default is **<logger>**.

Example

Example 23.15, "Logging configuration using annotations" shows how to enable logging for the **HelloWorld** SEI, where incoming messages are sent to **<stdout>** and outgoing messages are sent to **<logger>**.

Example 23.15. Logging configuration using annotations

```
@WebService
@Logging(limit=16000, inLocation="<stdout>")
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

23.2.3.6. Adding Properties and Policies to an Endpoint

Both properties and policies can be used to associate configuration data with an endpoint. The essential difference between them is that *properties* are a Apache CXF specific configuration mechanism whereas *policies* are a standard WSDL configuration mechanism. Policies typically originate from WS specifications and standards and they are normally set by defining **wsdl:policy** elements that appear in the WSDL contract. By contrast, properties are Apache CXF-specific and they are normally set by defining **jaxws:properties** elements in the Apache CXF Spring configuration file.

It is also possible, however, to define property settings and WSDL policy settings in Java using annotations, as described here.

23.2.3.6.1. Adding properties

@EndpointProperty annotation

The **@EndpointProperty** annotation is defined by the **org.apache.cxf.annotations.EndpointProperty** interface. It is placed on the SEI.

This annotation adds Apache CXF-specific configuration settings to an endpoint. Endpoint properties can also be specified in a Spring configuration file. For example, to configure WS-Security on an endpoint, you could add endpoint properties using the **jaxws:properties** element in a Spring

configuration file as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   ... >

   <jaxws:endpoint
      id="MyService"
      address="https://localhost:9001/MyService"
      serviceName="interop:MyService"
      endpointName="interop:MyServiceEndpoint"
      implementor="com.foo.MyService">

      <jaxws:properties>
         <entry key="ws-security.callback-handler"
value="interop.client.UTPasswordCallback"/>
         <entry key="ws-security.signature.properties"
value="etc/keystore.properties"/>
         <entry key="ws-security.encryption.properties"
value="etc/truststore.properties"/>
         <entry key="ws-security.encryption.username"
value="useReqSigCert"/>
      </jaxws:properties>

   </jaxws:endpoint>
</beans>
```

Alternatively, you could specify the preceding configuration settings in Java by adding **@EndpointProperty** annotations to the SEI, as shown in Example 23.16, “Configuring WS-Security Using @EndpointProperty Annotations”.

Example 23.16. Configuring WS-Security Using @EndpointProperty Annotations

```
@WebService
@EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback")
@EndpointProperty(name="ws-security.signature.properties"
value="etc/keystore.properties")
@EndpointProperty(name="ws-security.encryption.properties"
value="etc/truststore.properties")
@EndpointProperty(name="ws-security.encryption.username"
value="useReqSigCert")
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

@EndpointProperties annotation

The **@EndpointProperties** annotation is defined by the **org.apache.cxf.annotations.EndpointProperties** interface. It is placed on the SEI.

This annotation provides a way of grouping multiple **@EndpointProperty** annotations into a list.

Using **@EndpointProperties**, it is possible to re-write Example 23.16, "Configuring WS-Security Using @EndpointProperty Annotations" as shown in Example 23.17, "Configuring WS-Security Using an @EndpointProperties Annotation".

Example 23.17. Configuring WS-Security Using an @EndpointProperties Annotation

```
@WebService
@EndpointProperties(
  {
  @EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback"),
  @EndpointProperty(name="ws-security.signature.properties"
value="etc/keystore.properties"),
  @EndpointProperty(name="ws-security.encryption.properties"
value="etc/truststore.properties"),
  @EndpointProperty(name="ws-security.encryption.username"
value="useReqSigCert")
})
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

23.2.3.6.2. Adding policies

@Policy annotation

The **@Policy** annotation is defined by the **org.apache.cxf.annotations.Policy** interface. It can be placed on the SEI or the SEI methods.

This annotation is used to associate a WSDL policy with an SEI or an SEI method. The policy is specified by providing a URI that references an XML file containing a standard **wsdl:policy** element. If a WSDL contract is to be generated from the SEI (for example, using the **java2ws** command-line tool), you can specify whether or not you want to include this policy in the WSDL.

Table 23.13, "@Policy Properties" shows the properties supported by the **@Policy** annotation.

Table 23.13. @Policy Properties

Property	Description
uri	*(Required)* The location of the file containing the policy definition.
includeInWSDL	*(Optional)* Whether to include the policy in the generated contract, when generating WSDL. Default is **true**.
placement	*(Optional)* Specifies where in the WSDL file this documentation is to appear. For the list of possible placement values, see the section called "Placement in the WSDL contract".

Property	Description
faultClass	*(Optional)* If the placement is set to be **BINDING_OPERATION_FAULT** or **PORT_TYPE_OPERATION_FAULT**, you must also set this property to specify which fault this policy applies to. The value is the Java class that represents the fault.

@Policies annotation

The **@Policies** annotation is defined by the **org.apache.cxf.annotations.Policies** interface. It can be placed on the SEI or thse SEI methods.

This annotation provides a way of grouping multiple **@Policy** annotations into a list.

Placement in the WSDL contract

To specify where the policy should appear in the WSDL contract, you can specify the **placement** property, which is of type **Policy.Placement**. The placement can have one of the following values:

```
Policy.Placement.BINDING
Policy.Placement.BINDING_OPERATION
Policy.Placement.BINDING_OPERATION_FAULT
Policy.Placement.BINDING_OPERATION_INPUT
Policy.Placement.BINDING_OPERATION_OUTPUT
Policy.Placement.DEFAULT
Policy.Placement.PORT_TYPE
Policy.Placement.PORT_TYPE_OPERATION
Policy.Placement.PORT_TYPE_OPERATION_FAULT
Policy.Placement.PORT_TYPE_OPERATION_INPUT
Policy.Placement.PORT_TYPE_OPERATION_OUTPUT
Policy.Placement.SERVICE
Policy.Placement.SERVICE_PORT
```

Example of @Policy

The following example shows how to associate WSDL policies with the **HelloWorld** SEI and how to associate a policy with the **sayHi** method. The policies themselves are stored in XML files in the file system, under the **annotationpolicies** directory.

```
@WebService
@Policy(uri = "annotationpolicies/TestImplPolicy.xml",
        placement = Policy.Placement.SERVICE_PORT),
@Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
        placement = Policy.Placement.PORT_TYPE)
public interface HelloWorld {
    @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
            placement = Policy.Placement.PORT_TYPE_OPERATION),
    String sayHi(@WebParam(name = "text") String text);
}
```

Example of @Policies

You can use the **@Policies** annotation to group multiple **@Policy** annotations into a list, as shown in the following example:

```
@WebService
@Policies({
    @Policy(uri = "annotationpolicies/TestImplPolicy.xml",
            placement = Policy.Placement.SERVICE_PORT),
    @Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
            placement = Policy.Placement.PORT_TYPE)
})
public interface HelloWorld {
    @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
            placement = Policy.Placement.PORT_TYPE_OPERATION),
    String sayHi(@WebParam(name = "text") String text);
}
```

23.3. GENERATING WSDL

Using Maven

Once your code is annotated, you can generate a WSDL contract for your service using the **java2ws** Maven plug-in's **-wsdl** option. For a detailed listing of options for the **java2ws** Maven plug-in see java2ws.

Example 23.18, “Generating WSDL from Java” shows how to set up the **java2ws** Maven plug-in to generate WSDL.

Example 23.18. Generating WSDL from Java

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <genWsdl>true</genWsdl>
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Example

Example 23.19, “Generated WSDL from an SEI” shows the WSDL contract that is generated for the SEI shown in Example 23.7, “Fully Annotated SEI”.

Example 23.19. Generated WSDL from an SEI

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://demo.eric.org/"
          xmlns:tns="http://demo.eric.org/"
    xmlns:ns1=""
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns2="http://demo.eric.org/types"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema>
      <xs:complexType name="quote">
        <xs:sequence>
          <xs:element name="ID" type="xs:string" minOccurs="0"/>
          <xs:element name="time" type="xs:string" minOccurs="0"/>
          <xs:element name="val" type="xs:float"/>
        </xs:sequence>
      </xs:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="getStockQuote">
    <wsdl:part name="stockTicker" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getStockQuoteResponse">
    <wsdl:part name="updatedQuote" type="tns:quote">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="quoteReporter">
    <wsdl:operation name="getStockQuote">
      <wsdl:input name="getQuote" message="tns:getStockQuote">
    </wsdl:input>
      <wsdl:output name="getQuoteResponse"
message="tns:getStockQuoteResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="quoteReporterBinding" type="tns:quoteReporter">
    <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="getStockQuote">
      <soap:operation style="rpc" />
      <wsdl:input name="getQuote">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="getQuoteResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="quoteReporterService">
    <wsdl:port name="quoteReporterPort"
```

```
binding="tns:quoteReporterBinding">
      <soap:address
location="http://localhost:9000/quoteReporterService" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

[2] **Board** is an assumed class whose implementation is left to the reader.

CHAPTER 24. DEVELOPING A CONSUMER WITHOUT A WSDL CONTRACT

Abstract

You do not need a WSDL contract to develop a service consumer. You can create a service consumer from an annotated SEI. Along with the SEI you need to know the address at which the endpoint exposing the service is published, the QName of the service element that defines the endpoint exposing the service, and the QName of the port element defining the endpoint on which your consumer makes requests. This information can be specified in the SEI's annotations or provided separately.

To create a consumer without a WSDL contract you must do the following:

1. Create a `Service` object for the service on which the consumer will invoke operations.
2. Add a port to the `Service` object.
3. Get a proxy for the service using the `Service` object's `getPort()` method.
4. Implement the consumer's business logic.

24.1. CREATING A SERVICE OBJECT

Overview

The `javax.xml.ws.Service` class represents the `wsdl:service` element which contains the definition of all of the endpoints that expose a service. As such, it provides methods that allow you to get endpoints, defined by `wsdl:port` elements, that are proxies for making remote invocations on a service.

NOTE

The `Service` class provides the abstractions that allow the client code to work with Java types as opposed to working with XML documents.

The `create()` methods

The `Service` class has two static `create()` methods that can be used to create a new `Service` object. As shown in Example 24.1, "`Service create()` Methods", both of the `create()` methods take the QName of the `wsdl:service` element the `Service` object will represent, and one takes a URI specifying the location of the WSDL contract.

TIP

All services publish their WSDL contracts. For SOAP/HTTP services the URI is usually the URI for the service appended with `?wsdl`.

Example 24.1. `Service create()` Methods

```
public static Service create(URL wsdlLocation,
```

```
                              QName serviceName)
    throws WebServiceException;
public static Service create(QName serviceName)
    throws WebServiceException;
```

The value of the ***serviceName*** parameter is a QName. The value of its namespace part is the target namespace of the service. The service's target namespace is specified in the targetNamespace property of the **@WebService** annotation. The value of the QName's local part is the value of **wsdl:service** element's **name** attribute. You can determine this value in one of the following ways:

1. It is specified in the serviceName property of the **@WebService** annotation.
2. You append **Service** to the value of the name property of the **@WebService** annotation.
3. You append **Service** to the name of the SEI.

Example

Example 24.2, "Creating a **Service** Object" shows code for creating a **Service** object for the SEI shown in Example 23.7, "Fully Annotated SEI".

Example 24.2. Creating a Service Object

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
1    QName serviceName = new QName("http://demo.redhat.com",
  "stockQuoteReporter");
2    Service s = Service.create(serviceName);
     ...
  }
}
```

The code in Example 24.2, "Creating a **Service** Object" does the following:

1. Builds the QName for the service using the targetNamespace property and the name property of the **@WebService** annotation.
2. Calls the single parameter **create()** method to create a new **Service** object.

NOTE

Using the single parameter **create()** frees you from having any dependencies on accessing a WSDL contract.

24.2. ADDING A PORT TO A SERVICE

Overview

The endpoint information for a service is defined in a **wsdl:port** element, and the **Service** object creates a proxy instance for each of the endpoints defined in a WSDL contract, if one is specified. If you do not specify a WSDL contract when you create your **Service** object, the **Service** object has no information about the endpoints that implement your service, and therefore cannot create any proxy instances. In this case, you must provide the **Service** object with the information needed to represent a **wsdl:port** element using the **addPort()** method.

The addPort() method

The **Service** class defines an **addPort()** method, shown in Example 24.3, "The **addPort()** Method", that is used in cases where there is no WSDL contract available to the consumer implementation. The **addPort()** method allows you to give a **Service** object the information, which is typically stored in a **wsdl:port** element, necessary to create a proxy for a service implementation.

Example 24.3. The addPort() Method

```
void addPort(QName portName,
             String bindingId,
             String endpointAddress)
    throws WebServiceException;
```

The value of the ***portName*** is a QName. The value of its namespace part is the target namespace of the service. The service's target namespace is specified in the targetNamespace property of the **@WebService** annotation. The value of the QName's local part is the value of **wsdl:port** element's **name** attribute. You can determine this value in one of the following ways:

1. Specify it in the portName property of the **@WebService** annotation.
2. Append **Port** to the value of the name property of the **@WebService** annotation.
3. Append **Port** to the name of the SEI.

The value of the ***bindingId*** parameter is a string that uniquely identifies the type of binding used by the endpoint. For a SOAP binding you use the standard SOAP namespace: **http://schemas.xmlsoap.org/soap/**. If the endpoint is not using a SOAP binding, the value of the ***bindingId*** parameter is determined by the binding developer.

The value of the ***endpointAddress*** parameter is the address where the endpoint is published. For a SOAP/HTTP endpoint, the address is an HTTP address. Transports other than HTTP use different address schemes.

Example

Example 24.4, "Adding a Port to a **Service** Object" shows code for adding a port to the **Service** object created in Example 24.2, "Creating a **Service** Object".

Example 24.4. Adding a Port to a Service Object

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    ...
1     QName portName = new QName("http://demo.redhat.com",
  "stockQuoteReporterPort");
2     s.addPort(portName,
3                 "http://schemas.xmlsoap.org/soap/",
4                   "http://localhost:9000/StockQuote");
          ...
  }
}
```

The code in Example 24.4, "Adding a Port to a `Service` Object" does the following:

1. Creates the QName for the ***`portName`*** parameter.
2. Calls the **`addPort()`** method.
3. Specifies that the endpoint uses a SOAP binding.
4. Specifies the address where the endpoint is published.

24.3. GETTING A PROXY FOR AN ENDPOINT

Overview

A service proxy is an object that provides all of the methods exposed by a remote service and handles all of the details required to make the remote invocations. The **`Service`** object provides service proxies for all of the endpoints it is aware of through the **`getPort()`** method. Once you have a service proxy, you can invoke its methods. The proxy forwards the invocation to the remote service endpoint using the connection details specified in the service's contract.

The `getPort()` method

The **`getPort()`** method, shown in Example 24.5, "The `getPort()` Method", returns a service proxy for the specified endpoint. The returned proxy is of the same class as the SEI.

Example 24.5. The `getPort()` Method

```
public <T> T getPort(QName portName,
                     Class<T> serviceEndpointInterface)
    throws WebServiceException;
```

The value of the ***portName*** parameter is a QName that identifies the `wsdl:port` element that defines the endpoint for which the proxy is created. The value of the ***serviceEndpointInterface*** parameter is the fully qualified name of the SEI.

TIP

When you are working without a WSDL contract the value of the ***portName*** parameter is typically the same as the value used for the ***portName*** parameter when calling `addPort()`.

Example

Example 24.6, "Getting a Service Proxy" shows code for getting a service proxy for the endpoint added in Example 24.4, "Adding a Port to a `Service` Object".

Example 24.6. Getting a Service Proxy

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    ...
    quoteReporter proxy = s.getPort(portName, quoteReporter.class);
    ...
  }
}
```

24.4. IMPLEMENTING THE CONSUMER'S BUSINESS LOGIC

Overview

Once you instantiate a service proxy for a remote endpoint, you can invoke its methods as if it were a local object. The calls block until the remote method completes.

NOTE

If a method is annotated with the **@OneWay** annotation, the call returns immediately.

Example

Example 24.7, "Consumer Implemented without a WSDL Contract" shows a consumer for the service defined in Example 23.7, "Fully Annotated SEI".

Example 24.7. Consumer Implemented without a WSDL Contract

```
package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    QName serviceName = new QName("http://demo.eric.org",
"stockQuoteReporter");
1   Service s = Service.create(serviceName);

    QName portName = new QName("http://demo.eric.org",
"stockQuoteReporterPort");
2   s.addPort(portName, "http://schemas.xmlsoap.org/soap/",
   "http://localhost:9000/EricStockQuote");

3   quoteReporter proxy = s.getPort(portName, quoteReporter.class);

4   Quote quote = proxy.getQuote("ALPHA");
      System.out.println("Stock "+quote.getID()+" is worth
"+quote.getVal()+" as of "+quote.getTime());
  }
}
```

The code in Example 24.7, “Consumer Implemented without a WSDL Contract” does the following:

1. Creates a **`Service`** object.
2. Adds an endpoint definition to the **`Service`** object.
3. Gets a service proxy from the **`Service`** object.
4. Invokes an operation on the service proxy.

CHAPTER 25. A STARTING POINT WSDL CONTRACT

Example 25.1, "HelloWorld WSDL Contract" shows the HelloWorld WSDL contract. This contract defines a single interface, **Greeter**, in the **wsdl:portType** element. The contract also defines the endpoint which will implement the service in the **wsdl:port** element.

Example 25.1. HelloWorld WSDL Contract

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"

targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"

xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema
targetNamespace="http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"><element name="sayHi">
      <element name="sayHi">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="sayHiResponse">
        <complexType>
          <sequence>
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMe">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeResponse">
        <complexType>
          <sequence>
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeOneWay">
        <complexType>
          <sequence>
```

```
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="pingMe">
        <complexType/>
      </element>
      <element name="pingMeResponse">
        <complexType/>
      </element>
      <element name="faultDetail">
        <complexType>
          <sequence>
            <element name="minor" type="short"/>
            <element name="major" type="short"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>

  <wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
  </wsdl:message>
  <wsdl:message name="sayHiResponse">
    <wsdl:part element="x1:sayHiResponse" name="out"/>
  </wsdl:message>
  <wsdl:message name="greetMeRequest">
    <wsdl:part element="x1:greetMe" name="in"/>
  </wsdl:message>
  <wsdl:message name="greetMeResponse">
    <wsdl:part element="x1:greetMeResponse" name="out"/>
  </wsdl:message>
  <wsdl:message name="greetMeOneWayRequest">
    <wsdl:part element="x1:greetMeOneWay" name="in"/>
  </wsdl:message>
  <wsdl:message name="pingMeRequest">
    <wsdl:part name="in" element="x1:pingMe"/>
  </wsdl:message>
  <wsdl:message name="pingMeResponse">
    <wsdl:part name="out" element="x1:pingMeResponse"/>
  </wsdl:message>
  <wsdl:message name="pingMeFault">
    <wsdl:part name="faultDetail" element="x1:faultDetail"/>
  </wsdl:message>

  <wsdl:portType name="Greeter">
1     <wsdl:operation name="sayHi">
        <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
      <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
    </wsdl:operation>

2     <wsdl:operation name="greetMe">
        <wsdl:input message="tns:greetMeRequest"
name="greetMeRequest"/>
      <wsdl:output message="tns:greetMeResponse"
```

```
name="greetMeResponse"/>
    </wsdl:operation>

3    <wsdl:operation name="greetMeOneWay">
       <wsdl:input message="tns:greetMeOneWayRequest"
name="greetMeOneWayRequest"/>
    </wsdl:operation>

4    <wsdl:operation name="pingMe">
       <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
      <wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
      <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
    ...
  </wsdl:binding>

  <wsdl:service name="SOAPService">
    <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
      <soap:address
location="http://localhost:9000/SoapContext/SoapPort"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

The **Greeter** interface defined in Example 25.1, "HelloWorld WSDL Contract" defines the following operations:

1. sayHi — Has a single output parameter, of xsd:string.
2. greetMe — Has an input parameter, of xsd:string, and an output parameter, of xsd:string.
3. greetMeOneWay — Has a single input parameter, of xsd:string. Because this operation has no output parameters, it is optimized to be a oneway invocation (that is, the consumer does not wait for a response from the server).
4. pingMe — Has no input parameters and no output parameters, but it can raise a fault exception.

CHAPTER 26. TOP-DOWN SERVICE DEVELOPMENT

Abstract

In the top-down method of developing a service provider you start from a WSDL document that defines the operations and methods the service provider will implement. Using the WSDL document, you generate starting point code for the service provider. Adding the business logic to the generated code is done using normal Java programming APIs.

Once you have a WSDL document, the process for developing a JAX-WS service provider is as follows:

1. Generate starting point code.
2. Implement the service provider's operations.
3. Publish the implemented service.

26.1. GENERATING THE STARTING POINT CODE

Overview

JAX-WS specifies a detailed mapping from a service defined in WSDL to the Java classes that will implement that service as a service provider. The logical interface, defined by the **`wsdl:portType`** element, is mapped to a service endpoint interface (SEI). Any complex types defined in the WSDL are mapped into Java classes following the mapping defined by the Java Architecture for XML Binding (JAXB) specification. The endpoint defined by the **`wsdl:service`** element is also generated into a Java class that is used by consumers to access service providers implementing the service.

The **`cxf-codegen-plugin`** Maven plug-in generates this code. It also provides options for generating starting point code for your implementation. The code generator provides a number of options for controlling the generated code.

Running the code generator

Example 26.1, “Service Code Generation” shows how to use the code generator to generate starting point code for a service.

Example 26.1. Service Code Generation

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>wsdl</wsdl>
            <extraargs>
```

```
            <extraarg>-server</extraarg>
            <extraarg>-impl</extraarg>
          </extraargs>
        </wsdlOption>
      </wsdlOptions>
    </configuration>
    <goals>
      <goal>wsdl2java</goal>
    </goals>
  </execution>
 </executions>
</plugin>
```

This does the following:

- The **-impl** option generates a shell implementation class for each **wsdl:portType** element in the WSDL contract.
- The **-server** option generates a simple **main()** to run your service provider as a stand alone application.
- The **sourceRoot** specifies that the generated code is written to a directory called *outputDir*.
- **wsdl** element specifies the WSDL contract from which code is generated.

For a complete list of the options for the code generator see cxf-codegen-plugin.

Generated code

Table 26.1, "Generated Classes for a Service Provider" describes the files generated for creating a service provider.

Table 26.1. Generated Classes for a Service Provider

File	Description
***portTypeName*.java**	The SEI. This file contains the interface your service provider implements. You should not edit this file.
***serviceName*.java**	The endpoint. This file contains the Java class consumers use to make requests on the service.
***portTypeName*Impl.java**	The skeleton implementation class. Modify this file to build your service provider.
***portTypeName*Server.java**	A basic server mainline that allows you to deploy your service provider as a stand alone process. For more information see Chapter 30, *Publishing a Service*.

In addition, the code generator will generate Java classes for all of the types defined in the WSDL contract.

Generated packages

The generated code is placed into packages based on the namespaces used in the WSDL contract. The classes generated to support the service (based on the **wsdl:portType** element, the **wsdl:service** element, and the **wsdl:port** element) are placed in a package based on the target namespace of the WSDL contract. The classes generated to implement the types defined in the **types** element of the contract are placed in a package based on the **targetNamespace** attribute of the **types** element.

The mapping algorithm is as follows:

1. The leading **http://** or **urn://** are stripped off the namespace.
2. If the first string in the namespace is a valid Internet domain, for example it ends in **.com** or **.gov**, then the leading **www.** is stripped off the string, and the two remaining components are flipped.
3. If the final string in the namespace ends with a file extension of the pattern **.xxx** or **.xx**, then the extension is stripped.
4. The remaining strings in the namespace are appended to the resulting string and separated by dots.
5. All letters are made lowercase.

26.2. IMPLEMENTING THE SERVICE PROVIDER

Generating the implementation code

You generate the implementation class used to build your service provider with the code generator's **-impl** flag.

TIP

If your service's contract includes any custom types defined in XML Schema, you must ensure that the classes for the types are generated and available.

For more information on using the code generator see cxf-codegen-plugin.

Generated code

The implementation code consists of two files:

- ***portTypeName*.java** — The service interface(SEI) for the service.
- ***portTypeName*Impl.java** — The class you will use to implement the operations defined by the service.

Implement the operation's logic

To provide the business logic for your service's operations complete the stub methods in ***portTypeName*Impl.java**. You usually use standard Java to implement the business logic. If your service uses custom XML Schema types, you must use the generated classes for each type to

manipulate them. There are also some Apache CXF specific APIs that can be used to access some advanced features.

Example

For example, an implementation class for the service defined in Example 25.1, "HelloWorld WSDL Contract" may look like Example 26.2, "Implementation of the Greeter Service". Only the code portions highlighted in bold must be inserted by the programmer.

Example 26.2. Implementation of the Greeter Service

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName =
"SOAPService",
                      targetNamespace =
"http://apache.org/hello_world_soap_http",
                      endpointInterface =
"org.apache.hello_world_soap_http.Greeter")

public class GreeterImpl implements Greeter {

    public String greetMe(String me) {
       System.out.println("Executing operation greetMe");
       System.out.println("Message received: " + me + "\n");
       return "Hello " + me;
    }

    public void greetMeOneWay(String me) {
       System.out.println("Executing operation greetMeOneWay\n");
       System.out.println("Hello there " + me);
    }

    public String sayHi() {
       System.out.println("Executing operation sayHi\n");
       return "Bonjour";
    }

    public void pingMe() throws PingMeFault {
       FaultDetail faultDetail = new FaultDetail();
       faultDetail.setMajor((short)2);
       faultDetail.setMinor((short)1);
       System.out.println("Executing operation pingMe, throwing
PingMeFault exception\n");
       throw new PingMeFault("PingMeFault raised by server",
faultDetail);
    }
}
```

CHAPTER 27. DEVELOPING A CONSUMER FROM A WSDL CONTRACT

Abstract

One way method of creating a consumer is to start from a WSDL contract. The contract defines the operations, messages, and transport details of the service on which a consumer makes requests. The starting point code for the consumer is generated from the WSDL contract. The functionality required by the consumer is added to the generated code.

27.1. GENERATING THE STUB CODE

Overview

The **cxf-codegen-plugin** Maven plug-in generates the stub code from the WSDL contract. The stub code provides the supporting code that is required to invoke operations on the remote service.

For consumers, the **cxf-codegen-plugin** Maven plug-in generates the following types of code:

- Stub code — Supporting files for implementing a consumer.
- Starting point code — Sample code that connects to the remote service and invokes every operation on the remote service.

Generating the consumer code

To generate consumer code use the **cxf-codegen-plugin** Maven plug-in. Example 27.1, “Consumer Code Generation” shows how to use the code generator to generate consumer code.

Example 27.1. Consumer Code Generation

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>wsdl</wsdl>
            <extraargs>
              <extraarg>-client</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
```

```
        </goals>
      </execution>
    </executions>
  </plugin>
```

Where *outputDir* is the location of a directory where the generated files are placed and *wsdl* specifies the WSDL contract's location. The **-client** option generates starting point code for the consumer's **main()** method.

For a complete list of the arguments available for the **cxf-codegen-plugin** Maven plug-in see cxf-codegen-plugin.

Generated code

The code generation plug-in generates the following Java packages for the contract shown in Example 25.1, "HelloWorld WSDL Contract":

- org.apache.hello_world_soap_http — This package is generated from the **http://apache.org/hello_world_soap_http** target namespace. All of the WSDL entities defined in this namespace (for example, the Greeter port type and the SOAPService service) map to Java classes this Java package.
- org.apache.hello_world_soap_http.types — This package is generated from the **http://apache.org/hello_world_soap_http/types** target namespace. All of the XML types defined in this namespace (that is, everything defined in the **wsdl:types** element of the HelloWorld contract) map to Java classes in this Java package.

The stub files generated by the **cxf-codegen-plugin** Maven plug-in fall into the following categories:

- Classes representing WSDL entities in the org.apache.hello_world_soap_http package. The following classes are generated to represent WSDL entities:
  - **Greeter** — A Java interface that represents the Greeter **wsdl:portType** element. In JAX-WS terminology, this Java interface is the service endpoint interface (SEI).
  - **SOAPService** — A Java service class (extending **javax.xml.ws.Service**) that represents the SOAPService **wsdl:service** element.
  - **PingMeFault** — A Java exception class (extending **java.lang.Exception**) that represents the pingMeFault **wsdl:fault** element.
- Classes representing XML types in the org.objectweb.hello_world_soap_http.types package. In the HelloWorld example, the only generated types are the various wrappers for the request and reply messages. Some of these data types are useful for the asynchronous invocation model.

27.2. IMPLEMENTING A CONSUMER

Overview

To implement a consumer when starting from a WSDL contract, you must use the following stubs:

- Service class
- SEI

Using these stubs, the consumer code instantiates a service proxy to make requests on the remote service. It also implements the consumer's business logic.

Generated service class

Example 27.2, "Outline of a Generated Service Class" shows the typical outline of a generated service class, ***ServiceName*_Service**[3], which extends the **javax.xml.ws.Service** base class.

Example 27.2. Outline of a Generated Service Class

```
@WebServiceClient(name="..." targetNamespace="..."
                  wsdlLocation="...")
public class ServiceName extends javax.xml.ws.Service
{
  ...
  public ServiceName(URL wsdlLocation, QName serviceName) { }

  public ServiceName() { }

  @WebEndpoint(name="...")
  public SEI getPortName() { }
  .
  .
  .
}
```

The ***ServiceName*** class in Example 27.2, "Outline of a Generated Service Class" defines the following methods:

- ***ServiceName*(URL wsdlLocation, QName serviceName)** — Constructs a service object based on the data in the **wsdl:service** element with the QName *ServiceName* service in the WSDL contract that is obtainable from *wsdlLocation*.
- ***ServiceName*()** — The default constructor. It constructs a service object based on the service name and the WSDL contract that were provided at the time the stub code was generated (for example, when running the **wsdl2java** tool). Using this constructor presupposes that the WSDL contract remains available at a specified location.
- **get*PortName*()** — Returns a proxy for the endpoint defined by the **wsdl:port** element with the **name** attribute equal to *PortName*. A getter method is generated for every **wsdl:port** element defined by the *ServiceName* service. A **wsdl:service** element that contains multiple endpoint definitions results in a generated service class with multiple **get*PortName*()** methods.

Service endpoint interface

For every interface defined in the original WSDL contract, you can generate a corresponding SEI. A service endpoint interface is the Java mapping of a **wsdl:portType** element. Each operation defined in the original **wsdl:portType** element maps to a corresponding method in the SEI. The operation's

parameters are mapped as follows:

1. The input parameters are mapped to method arguments.
2. The first output parameter is mapped to a return value.
3. If there is more than one output parameter, the second and subsequent output parameters map to method arguments (moreover, the values of these arguments must be passed using Holder types).

For example, Example 27.3, "The Greeter Service Endpoint Interface" shows the Greeter SEI, which is generated from the **wsdl:portType** element defined in Example 25.1, "HelloWorld WSDL Contract". For simplicity, Example 27.3, "The Greeter Service Endpoint Interface" omits the standard JAXB and JAX-WS annotations.

Example 27.3. The Greeter Service Endpoint Interface

```
package org.apache.hello_world_soap_http;
  ...
public interface Greeter
{
  public String sayHi();
  public String greetMe(String requestType);
  public void greetMeOneWay(String requestType);
  public void pingMe() throws PingMeFault;
}
```

Consumer main function

Example 27.4, "Consumer Implementation Code" shows the code that implements the HelloWorld consumer. The consumer connects to the SoapPort port on the SOAPService service and then proceeds to invoke each of the operations supported by the Greeter port type.

Example 27.4. Consumer Implementation Code

```
package demo.hw.client;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.SOAPService;

public final class Client {

  private static final QName SERVICE_NAME =
  new QName("http://apache.org/hello_world_soap_http",
            "SOAPService");

  private Client()
  {
  }
```

```
  public static void main(String args[]) throws Exception
  {
1 if (args.length == 0)
     {
      System.out.println("please specify wsdl");
      System.exit(1);
    }

2 URL wsdlURL;
     File wsdlFile = new File(args[0]);
    if (wsdlFile.exists())
    {
      wsdlURL = wsdlFile.toURL();
    }
    else
    {
      wsdlURL = new URL(args[0]);
    }

    System.out.println(wsdlURL);
3 SOAPService ss = new SOAPService(wsdlURL,SERVICE_NAME);
  4 Greeter port = ss.getSoapPort();
       String resp;

5 System.out.println("Invoking sayHi...");
     resp = port.sayHi();
    System.out.println("Server responded with: " + resp);
    System.out.println();

    System.out.println("Invoking greetMe...");
    resp = port.greetMe(System.getProperty("user.name"));
    System.out.println("Server responded with: " + resp);
    System.out.println();

    System.out.println("Invoking greetMeOneWay...");
    port.greetMeOneWay(System.getProperty("user.name"));
    System.out.println("No response from server as method is OneWay");
    System.out.println();

6 try {
       System.out.println("Invoking pingMe, expecting exception...");
      port.pingMe();
    } catch (PingMeFault ex) {
      System.out.println("Expected exception: PingMeFault has
occurred.");
      System.out.println(ex.toString());
    }
    System.exit(0);
  }
}
```

The **`Client.main()`** method from Example 27.4, “Consumer Implementation Code” proceeds as follows:

1. Provided that the Apache CXF runtime classes are on your classpath, the runtime is implicitly initialized. There is no need to call a special function to initialize Apache CXF.

2. The consumer expects a single string argument that gives the location of the WSDL contract for HelloWorld. The WSDL contract's location is stored in **`wsdlURL`**.

3. You create a service object using the constructor that requires the WSDL contract's location and service name.

4. Call the appropriate **`get`*`PortName`*`()`** method to obtain an instance of the required port. In this case, the SOAPService service supports only the SoapPort port, which implements the **`Greeter`** service endpoint interface.

5. The consumer invokes each of the methods supported by the **`Greeter`** service endpoint interface.

6. In the case of the **`pingMe()`** method, the example code shows how to catch the **`PingMeFault`** fault exception.

---

[3] If the **`name`** attribute of the **`wsdl:service`** element ends in **`Service`** the **`_Service`** is not used.

CHAPTER 28. FINDING WSDL AT RUNTIME

Abstract

Hard coding the location of WSDL documents into an application is not scalable. In real deployment environments, you will want to allow the WSDL document's location be resolved at runtime. Apache CXF provides a number of tools to make this possible.

When developing consumers using the JAX-WS APIs you are must provide a hard coded path to the WSDL document that defines your service. While this is OK in a small environment, using hard coded paths does not translate to enterprise deployments.

To address this issue, Apache CXF provides three mechanisms for removing the requirement of using hard coded paths:

- inject a configured proxy object
- a JAX-WS catalog
- the

TIP

Injecting the proxy into your implementation code is generally the best option.

28.1. INSTANTIATING A PROXY BY INJECTION

Overview

Apache CXF's use of the Spring Framework allows you to avoid the hassle of using the JAX-WS APIs to create service proxies. It allows you to define a client endpoint in a configuration file and then inject a proxy directly into the implementation code. When the runtime instantiates the implementation object, it will also instantiate a proxy for the external service based on the configuration. The implementation is handed a reference to the instantiated proxy.

Because the proxy is instantiated using information in the configuration file, the WSDL location does not need to be hard coded. It can be changed at deployment time. You can also specify that the runtime should search the application's classpath for the WSDL.

Procedure

To inject a proxy for an external service into a service provider's implementation do the following:

1. Deploy the required WSDL documents in a well known location that all parts of the application can access.

   TIP

   If you are deploying the application as a WAR file, it is recommended that you place all of the WSDL documents and XML Schema documents in the `WEB-INF/wsdl` folder of the WAR.

TIP

If you are deploying the application as a JAR file, it is recommended that you place all of the WSDL documents and XML Schema documents in the **`META-INF/wsdl`** folder of the JAR.

2. Configure a JAX-WS client endpoint for the proxy that is being injected.
3. Inject the proxy into your service provide using the **`@Resource`** annotation.

Configuring the proxy

You configure a JAX-WS client endpoint using the **`jaxws:client`** element in you application's configuration file. This tells the runtime to instantiate a **`org.apache.cxf.jaxws.JaxWsClientProxy`** object with the specified properties. This object is the proxy that will be injected into the service provider.

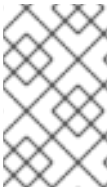
At a minimum you need to provide values for the following attributes:

- **`id`**—Specifies the ID used to identify the client to be injected.
- **`serviceClass`**—Specifies the SEI of the service on which the proxy makes requests.

Example 28.1, "Configuration for a Proxy to be Injected into a Service Implementation" shows the configuration for a JAX-WS client endpoint.

Example 28.1. Configuration for a Proxy to be Injected into a Service Implementation

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:client id="bookClient"
                serviceClass="org.apache.cxf.demo.BookService"
                wsdlLocation="classpath:books.wsdl"/>
  ...
</beans>
```



NOTE

In Example 28.1, "Configuration for a Proxy to be Injected into a Service Implementation" the **`wsdlLocation`** attribute instructs the runtime to load the WSDL from the classpath. If **`books.wsdl`** in on the classpath, the runtime will be able to find it.

For more information on configuring a JAX-WS client see Section 15.2, "Configuring Consumer Endpoints".

Coding the provider implementation

You inject the configured proxy into a service implementation as a resource using the **`@Resource`** as shown in Example 28.2, "Injecting a Proxy into a Service Implementation".

Example 28.2. Injecting a Proxy into a Service Implementation

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName =
"SOAPService",
                      targetNamespace =
"http://apache.org/hello_world_soap_http",
                      endpointInterface =
"org.apache.hello_world_soap_http.Greeter")
public class StoreImpl implements Store {

@Resource(name="bookClient")
private BookService proxy;

}
```

The annotation's name property corresponds to the value of the JAX-WS client's **id** attribute. The configured proxy is injected into the **BookService** object declared immediately after the annotation. You can use this object to make invocations on the proxy's external service.

28.2. USING A JAX-WS CATALOG

Overview

The JAX-WS specification mandates the all implementations support:

> a standard catalog facility to be used when resolving any Web service document that is part of the description of a Web service, specifically WSDL and XML Schema documents.

This catalog facility uses the XML catalog facility specified by OASIS. All of the JAX-WS APIs and annotation that take a WSDL URI use the catalog to resolve the WSDL document's location.

This means that you can provide an XML catalog file that rewrites the locations of your WSDL documents to suite specific deployment environments.

Writing the catalog

JAX-WS catalogs are standard XML catalogs as defined by the OASIS XML Catalogs 1.1 specification. They allow you to specify mapping:

- a document's public identifier and/or a system identifier to a URI.
- the URI of a resource to another URI.

Table 28.1, “Common JAX-WS Catalog Elements” lists some common elements used for WSDL location resolution.

Table 28.1. Common JAX-WS Catalog Elements

Element	Description
uri	Maps a URI to an alternate URI.
rewriteURI	Rewrites the beginning of a URI. For example, this element allows you to map all URIs that start with http://cxf.apache.org to URIs that start with classpath:.
uriSuffix	Maps a URI to an alternate URI based on the suffix of the original URI. For example you could map all URIs that end in foo.xsd to classpath:foo.xsd.

Packaging the catalog

The JAX-WS specification mandates that the catalog used to resolve WSDL and XML Schema documents is assembled using all available resources named **META-INF/jax-ws-catalog.xml**. If your application is packaged into a single JAR, or WAR, you can place the catalog into a single file.

If your application is packaged as multiple JARs, you can split the catalog into a number of files. Each catalog file could be modularized to only deal with WSDLs accessed by the code in the specific JARs.

28.3. USING A CONTRACT RESOLVER

Overview

The most involved mechanism for resolving WSDL document locations at runtime is to implement your own custom contract resolver. This requires that you provide an implementation of the Apache CXF specific **ServiceContractResolver** interface. You also need to register your custom resolver with the bus.

Once properly registered, the custom contract resolver will be used to resolve the location of any required WSDL and schema documents.

Implementing the contract resolver

A contract resolver is an implementation of the **org.apache.cxf.endpoint.ServiceContractResolver** interface. As shown in Example 28.3, "**ServiceContractResolver** Interface", this interface has a single method, **getContractLocation()**, that needs to be implemented. **getContractLocation()** takes the QName of a service and returns the URI for the service's WSDL contract.

Example 28.3. ServiceContractResolver Interface

```
public interface ServiceContractResolver
{
   URI getContractLocation(QName qname);
}
```

The logic used to resolve the WSDL contract's location is application specific. You can add logic that to resolve contract locations from a UDDI registry, a database, a custom location on a file system, or any other mechanism you choose.

Registering the contract resolver programmatically

Before the Apache CXF runtime will use your contract resolver, you must register it with a contract resolver registry. Contract resolver registries implement the **`org.apache.cxf.endpoint.ServiceContractResolverRegistry`** interface. However, you do not need to implement your own registry. Apache CXF provides a default implementation in the **`org.apache.cxf.endpoint.ServiceContractResolverRegistryImpl`** class.

To register a contract resolver with the default registry you do the following:

1. Get a reference to the default bus object.
2. Get the service contract registry from the bus using the bus' **`getExtension()`** method.
3. Create an instance of your contract resolver.
4. Register your contract resolver with the registry using the registry's **`register()`** method.

Example 28.4, “Registering a Contract Resolver” shows the code for registering a contract resolver with the default registry.

Example 28.4. Registering a Contract Resolver

```
BusFactory bf=BusFactory.newInstance();
Bus bus=bf.createBus();

ServiceContractResolverRegistry registry =
bus.getExtension(ServiceContractResolverRegistry);

JarServiceContractResolver resolver = new JarServiceContractResolver();

registry.register(resolver);
```

The code in Example 28.4, “Registering a Contract Resolver” does the following:

1. Gets a bus instance.
2. Gets the bus' contract resolver registry.
3. Creates an instance of a contract resolver.
4. Registers the contract resolver with the registry.

Registering a contract resolver using configuration

You can also implement a contract resolver so that it can be added to a client through configuration. The contract resolver is implemented in such a way that when the runtime reads the configuration and instantiates the resolver, the resolver registers itself. Because the runtime handles the initialization, you can decide at runtime if a client needs to use the contract resolver.

To implement a contract resolver so that it can be added to a client through configuration do the following:

1. Add an **`init()`** method to your contract resolver implementation.
2. Add logic to your **`init()`** method that registers the contract resolver with the contract resolver registry as shown in Example 28.4, "Registering a Contract Resolver".
3. Decorate the **`init()`** method with the **`@PostConstruct`** annotation.

Example 28.5, "Service Contract Resolver that can be Registered Using Configuration" shows a contract resolver implementation that can be added to a client using configuration.

Example 28.5. Service Contract Resolver that can be Registered Using Configuration

```
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.xml.namespace.QName;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;

public class UddiResolver implements ServiceContractResolver
{
  private Bus bus;
  ...

  @PostConstruct
  public void init()
  {
    BusFactory bf=BusFactory.newInstance();
    Bus bus=bf.createBus();
    if (null != bus)
    {
      ServiceContractResolverRegistry resolverRegistry =
bus.getExtension(ServiceContractResolverRegistry.class);
      if (resolverRegistry != null)
      {
        resolverRegistry.register(this);
      }
    }
  }

  public URI getContractLocation(QName serviceName)
  {
    ...
  }
}
```

To register the contract resolver with a client you need to add a **bean** element to the client's configuration. The **bean** element's **`class`** attribute is the name of the class implementing the contract resolver.

Example 28.6, “Bean Configuring a Contract Resolver” shows a bean for adding a configuration resolver implemented by the **`org.apache.cxf.demos.myContractResolver`** class.

Example 28.6. Bean Configuring a Contract Resolver

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  ...
  <bean id="myResolver" class="org.apache.cxf.demos.myContractResolver"
/>
  ...
</beans>
```

Contract resolution order

When a new proxy is created, the runtime uses the contract registry resolver to locate the remote service's WSDL contract. The contract resolver registry calls each contract resolver's **`getContractLocation()`** method in the order in which the resolvers were registered. It returns the first URI returned from one of the registered contract resolvers.

If you registered a contract resolver that attempted to resolve the WSDL contract at a well known shared file system, it would be the only contract resolver used. However, if you subsequently registered a contract resolver that resolved WSDL locations using a UDDI registry, the registry could use both resolvers to locate a service's WSDL contract. The registry would first attempt to locate the contract using the shared file system contract resolver. If that contract resolver failed, the registry would then attempt to locate it using the UDDI contract resolver.

CHAPTER 29. GENERIC FAULT HANDLING

Abstract

The JAX-WS specification defines two type of faults. One is a generic JAX-WS runtime exception. The other is a protocol specific class of exceptions that is thrown during message processing.

29.1. RUNTIME FAULTS

Overview

Most of the JAX-WS APIs throw a generic **`javax.xml.ws.WebServiceException`** exception.

APIs that throw `WebServiceException`

Table 29.1, "APIs that Throw **`WebServiceException`**" lists some of the JAX-WS APIs that can throw the generic **`WebServiceException`** exception.

Table 29.1. APIs that Throw `WebServiceException`

API	Reason
`Binding.setHandlerChain()`	There is an error in the handler chain configuration.
`BindingProvider.getEndpointReference()`	The specified class is not assigned from a **`W3CEndpointReference`**.
`Dispatch.invoke()`	There is an error in the **`Dispatch`** instance's configuration or an error occurred while communicating with the service.
`Dispatch.invokeAsync()`	There is an error in the **`Dispatch`** instance's configuration.
`Dispatch.invokeOneWay()`	There is an error in the **`Dispatch`** instance's configuration or an error occurred while communicating with the service.
`LogicalMessage.getPayload()`	An error occurred when using a supplied **`JAXBContext`** to unmarshall the payload. The **`cause`** field of the **`WebServiceException`** contains the original **`JAXBException`**.
`LogicalMessage.setPayload()`	An error occurred when setting the payload of the message. If the exception is thrown when using a **`JAXBContext`**, the **`cause`** field of the **`WebServiceException`** contains the original **`JAXBException`**.

API	Reason
`WebServiceContext.getEndpointReference()`	The specified class is not assigned from a `W3CEndpointReference`.

29.2. PROTOCOL FAULTS

Overview

Protocol exceptions are thrown when an error occurs during the processing of a request. All synchronous remote invocations can throw a protocol exception. The underlying cause occurs either in the consumer's message handling chain or in the service provider.

The JAX-WS specification defines a generic protocol exception. It also specifies a SOAP-specific protocol exception and an HTTP-specific protocol exception.

Types of protocol exceptions

The JAX-WS specification defines three types of protocol exception. Which exception you catch depends on the transport and binding used by your application.

Table 29.2, "Types of Generic Protocol Exceptions" describes the three types of protocol exception and when they are thrown.

Table 29.2. Types of Generic Protocol Exceptions

Exception Class	When Thrown
`javax.xml.ws.ProtocolException`	This exception is the generic protocol exception. It can be caught regardless of the protocol in use. It can be cast into a specific fault type if you are using the SOAP binding or the HTTP binding. When using the XML binding in combination with the HTTP or JMS transports, the generic protocol exception cannot be cast into a more specific fault type.
`javax.xml.ws.soap.SOAPFaultException`	This exception is thrown by remote invocations when using the SOAP binding. For more information see the section called "Using the SOAP protocol exception".
javax.xml.ws.http.HTTPException	This exception is thrown when using the Apache CXF HTTP binding to develop RESTful Web services. For more information see Part VI, "Developing RESTful Web Services".

Using the SOAP protocol exception

The `SOAPFaultException` exception wraps a SOAP fault. The underlying SOAP fault is stored in the `fault` field as a `javax.xml.soap.SOAPFault` object.

If a service implementation needs to throw an exception that does not fit any of the custom exceptions created for the application, it can wrap the fault in a **SOAPFaultException** using the exceptions creator and throw it back to the consumer. Example 29.1, "Throwing a SOAP Protocol Exception" shows code for creating and throwing a **SOAPFaultException** if the method is passed an invalid parameter.

Example 29.1. Throwing a SOAP Protocol Exception

```
public Quote getQuote(String ticker)
{
  ...
  if(tickers.length()<3)
  {
    SOAPFault fault = SOAPFactory.newInstance().createFault();
    fault.setFaultString("Ticker too short");
    throw new SOAPFaultException(fault);
  }
  ...
}
```

When a consumer catches a **SOAPFaultException** exception they can retrieve the underlying cause of the exception by examining the wrapped **SOAPFault** exception. As shown in Example 29.2, "Getting the Fault from a SOAP Protocol Exception", the **SOAPFault** exception is retrieved using the **SOAPFaultException** exception's **getFault()** method.

Example 29.2. Getting the Fault from a SOAP Protocol Exception

```
...
try
{
  proxy.getQuote(ticker);
}
catch (SOAPFaultException sfe)
{
  SOAPFault fault = sfe.getFault();
  ...
}
```

CHAPTER 30. PUBLISHING A SERVICE

Abstract

When you want to deploy a JAX-WS service as a standalone Java application or in an OSGi container without Spring-DM, you must to implement the code that publishes the service provider.

Apache CXF provides a number of ways to publish a service as a service provider. How you publish a service depends on the deployment environment you are using. Many of the containers supported by Apache CXF do not require writing logic for publishing endpoints. There are two exceptions:

- deploying a server as a standalone Java application
- deploying a server into an OSGi container without Spring-DM

For detailed information in deploying applications into the supported containers see Part IV, "Configuring Web Service Endpoints".

30.1. APIS USED TO PUBLISH A SERVICE

Overview

The `javax.xml.ws.Enddpoint` class does the work of publishing a JAX-WS service provider. To publishing an endpoint do the following:

1. Create an `Endpoint` object for your service provider.
2. Publish the endpoint.
3. Stop the endpoint when application shuts down.

The `Endpoint` class provides methods for creating and publishing service providers. It also provides a method that can create and publish a service provider in a single method call.

Instantiating an service provider

A service provider is instantiated using an `Endpoint` object. You instantiate an `Endpoint` object for your service provider using one of the following methods:

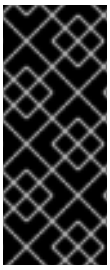
- `static Endpoint create(Object implementor);`
  This `create()` method returns an `Endpoint` for the specified service implementation. The `Endpoint` object is created using the information provided by the implementation class' `javax.xml.ws.BindingType` annotation, if it is present. If the annotation is not present, the `Endpoint` uses a default SOAP 1.1/HTTP binding.

- `static Endpoint create(URI bindingID, Object implementor);`
  This `create()` method returns an `Endpoint` object for the specified implementation object using the specified binding. This method overrides the binding information provided by the `javax.xml.ws.BindingType` annotation, if it is present. If the ***`bindingID`*** cannot be resolved, or it is `null`, the binding specified in the `javax.xml.ws.BindingType` is used to create the `Endpoint`. If neither the ***`bindingID`*** or the `javax.xml.ws.BindingType` can be used, the `Endpoint` is created using a default SOAP 1.1/HTTP binding.

- `static Endpoint publish(String address, Object implementor);`
  The `publish()` method creates an `Endpoint` object for the specified implementation, and publishes it. The binding used for the `Endpoint` object is determined by the URL scheme of the provided ***address***. The list of bindings available to the implementation are scanned for a binding that supports the URL scheme. If one is found the `Endpoint` object is created and published. If one is not found, the method fails.

  TIP

  Using `publish()` is the same as invoking one of the `create()` methods, and then invoking the `publish()` method used in publish to an address.



IMPORTANT

The implementation object passed to any of the `Endpoint` creation methods must either be an instance of a class annotated with `javax.jws.WebService` and meeting the requirements for being an SEI implementation or it must be an instance of a class annotated with `javax.xml.ws.WebServiceProvider` and implementing the `Provider` interface.

Publishing a service provider

You can publish a service provider using either of the following `Endpoint` methods:

- `void publish(String address);`
  This `publish()` method publishes the service provider at the address specified.

  IMPORTANT

  The ***address***'s URL scheme must be compatible with one of the service provider's bindings.

- `void publish(Object serverContext);`
  This `publish()` method publishes the service provider based on the information provided in the specified server context. The server context must define an address for the endpoint, and the context must also be compatible with one of the service provider's available bindings.

Stopping a published service provider

When the service provider is no longer needed you should stop it using its `stop()` method. The `stop()` method, shown in Example 30.1, “Method for Stopping a Published Endpoint” , shuts down the endpoint and cleans up any resources it is using.

Example 30.1. Method for Stopping a Published Endpoint

```
void stop();
```

IMPORTANT

Once the endpoint is stopped it cannot be republished.

30.2. PUBLISHING A SERVICE IN A PLAIN JAVA APPLICATION

Overview

When you want to deploy your application as a plain java application you need to implement the logic for publishing your endpoints in the application's **`main()`** method. Apache CXF provides you two options for writing your application's **`main()`** method.

- use the **`main()`** method generated by the **`wsdl2java`** tool
- write a custom **`main()`** method that publishes the endpoints

Generating a Server Mainline

The code generators **`-server`** flag makes the tool generate a simple server mainline. The generated server mainline, as shown in Example 30.2, "Generated Server Mainline", publishes one service provider for each **`port`** element in the specified WSDL contract.

For more information see cxf-codegen-plugin.

Example 30.2, "Generated Server Mainline" shows a generated server mainline.

Example 30.2. Generated Server Mainline

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer {

    protected GreeterServer() throws Exception {
        System.out.println("Starting Server");
1         Object implementor = new GreeterImpl();
2           String address =
   "http://localhost:9000/SoapContext/SoapPort";
3         Endpoint.publish(address, implementor);
      }

    public static void main(String args[]) throws Exception {
        new GreeterServer();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

The code in Example 30.2, "Generated Server Mainline" does the following:

1. Instantiates a copy of the service implementation object.
2.

Creates the address for the endpoint based on the contents of the **address** child of the **wsdl:port** element in the endpoint's contract.

3 Publishes the endpoint.

Writing a Server Mainline

If you used the Java first development model or you do not want to use the generated server mainline you can write your own. To write your server mainline you must do the following:

1. Instantiate an **javax.xml.ws.Endpoint** object for the service provider.
2. Create an optional server context to use when publishing the service provider.
3. Publish the service provider using one of the **publish()** methods.
4. Stop the service provider when the application is ready to exit.

Example 30.3, "Custom Server Mainline" shows the code for publishing a service provider.

Example 30.3. Custom Server Mainline

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer
{
  protected GreeterServer() throws Exception
  {
  }

  public static void main(String args[]) throws Exception
  {
1     GreeterImpl impl = new GreeterImpl();
2       Endpoint endpt.create(impl);
3         endpt.publish("http://localhost:9000/SoapContext/SoapPort");

    boolean done = false;
4    while(!done)
      {
      ...
    }

5    endpt.stop();
      System.exit(0);
  }
}
```

The code in Example 30.3, "Custom Server Mainline" does the following:

1 Instantiates a copy of the service's implementation object.

2. Creates an unpublished **Endpoint** for the service implementation.
3. Publishes the service provider at http://localhost:9000/SoapContext/SoapPort.
4. Loops until the server should be shutdown.
5. Stops the published endpoint.

30.3. PUBLISHING A SERVICE IN AN OSGI CONTAINER

Overview

When you develop an application that will be deployed into an OSGi container, you need to coordinate the publishing and stopping of your endpoints with the life-cycle of the bundle in which it is packaged. You want your endpoints published when the bundle is started and you want the endpoints stopped when the bundle is stopped.

You tie your endpoints life-cycle to the bundle's life-cycle by implementing an OSGi bundle activator. A bundle activator is used by the OSGi container to create the resource for a bundle when it is started. The container also uses the bundle activator to clean up the bundles resources when it is stopped.

The bundle activator interface

You create a bundle activator for your application by implementing the **org.osgi.framework.BundleActivator** interface. The **BundleActivator** interface, shown in Example 30.4, "Bundle Activator Interface", it has two methods that need to be implemented.

Example 30.4. Bundle Activator Interface

```
interface BundleActivator
{
  public void start(BundleContext context)
  throws java.lang.Exception;

  public void stop(BundleContext context)
  throws java.lang.Exception;
}
```

The **start()** method is called by the container when it starts the bundle. This is where you instantiate and publish the endpoints.

The **stop()** method is called by the container when it stops the bundle. This is where you would stop the endpoints.

Implementing the start method

The bundle activator's start method is where you publish your endpoints. To publish your endpoints the start method must do the following:

1. Instantiate an **javax.xml.ws.Endpoint** object for the service provider.
2. Create an optional server context to use when publishing the service provider.

3. Publish the service provider using one of the **`publish()`** methods.

Example 30.5, “Bundle Activator Start Method for Publishing an Endpoint” shows code for publishing a service provider.

Example 30.5. Bundle Activator Start Method for Publishing an Endpoint

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void start(BundleContext context)
  {
    WidgetOrderImpl impl = new WidgetOrderImpl(); (1)
    endpt = Endpoint.create(impl); (2)
    endpt.publish("http://localhost:9000/SoapContext/SoapPort"); (3)
  }

  ...

}
```

The code in Example 30.5, “Bundle Activator Start Method for Publishing an Endpoint” does the following:

1. Instantiates a copy of the service's implementation object.
2. Creates an unpublished **`Endpoint`** for the service implementation.
3. Publish the service provider at http://localhost:9000/SoapContext/SoapPort.

Implementing the stop method

The bundle activator's stop method is where you clean up the resources used by your application. Its implementation should include logic for stopping all of the endpoint's published by the application.

Example 30.6, “Bundle Activator Stop Method for Stopping an Endpoint” shows a stop method for stopping a published endpoint.

Example 30.6. Bundle Activator Stop Method for Stopping an Endpoint

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
```

```
public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void stop(BundleContext context)
  {
    endpt.stop();
  }

  ...

}
```

Informing the container

You must add inform the container that the application's bundle includes a bundle activator. You do this by adding the Bundle-Activator property to the bundle's manifest. This property tells the container which class in the bundle to use when activating the bundle. Its value is the fully qualified name of the class implementing the bundle activator.

Example 30.7, “Bundle Activator Manifest Entry” shows a manifest entry for a bundle whose activator is implemented by the class **`com.widgetvendor.osgi.widgetActivator`**.

Example 30.7. Bundle Activator Manifest Entry

```
Bundle-Activator: com.widgetvendor.osgi.widgetActivator
```

CHAPTER 31. BASIC DATA BINDING CONCEPTS

Abstract

There are a number of general topics that apply to how Apache CXF handles type mapping.

31.1. INCLUDING AND IMPORTING SCHEMA DEFINITIONS

Overview

Apache CXF supports the including and importing of schema definitions, using the **<include/>** and **<import/>** schema tags. These tags enable you to insert definitions from external files or resources into the scope of a schema element. The essential difference between including and importing is:

- Including brings in definitions that belong to the same target namespace as the enclosing schema element.
- Importing brings in definitions that belong to a different target namespace from the enclosing schema element.

xsd:include syntax

The include directive has the following syntax:

```
<include schemaLocation="anyURI" />
```

The referenced schema, given by *anyURI*, must either belong to the same target namespace as the enclosing schema, or not belong to any target namespace at all. If the referenced schema does not belong to any target namespace, it is automatically adopted into the enclosing schema's namespace when it is included.

Example 31.1, "Example of a Schema that Includes Another Schema" shows an example of an XML Schema document that includes another XML Schema document.

Example 31.1. Example of a Schema that Includes Another Schema

```
<definitions
targetNamespace="http://schemas.redhat.com/tests/schema_parser"
             xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema
targetNamespace="http://schemas.redhat.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <include schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="tns:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
```

```
  </types>
  ...
</definitions>
```

Example 31.2, “Example of an Included Schema” shows the contents of the included schema file.

Example 31.2. Example of an Included Schema

```
<schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

xsd:import syntax

The import directive has the following syntax:

```
<import namespace="namespaceAnyURI"
        schemaLocation="schemaAnyURI" />
```

The imported definitions must belong to the *namespaceAnyURI target* namespace. If *namespaceAnyURI* is blank or remains unspecified, the imported schema definitions are unqualified.

Example 31.3, “Example of a Schema that Includes Another Schema” shows an example of an XML Schema that imports another XML Schema.

Example 31.3. Example of a Schema that Includes Another Schema

```
<definitions
targetNamespace="http://schemas.redhat.com/tests/schema_parser"
             xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema
targetNamespace="http://schemas.redhat.com/tests/schema_parser"
            xmlns="http://www.w3.org/2001/XMLSchema">
      <import
namespace="http://schemas.redhat.com/tests/imported_types"
              schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="tns:IncludedSequence"/>
        </sequence>
      </complexType>
```

```
    </schema>
  </types>
  ...
</definitions>
```

Example 31.4, “Example of an Included Schema” shows the contents of the imported schema file.

Example 31.4. Example of an Included Schema

```
<schema targetNamespace="http://schemas.redhat.com/tests/imported_types"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

Using non-referenced schema documents

Using types defined in a schema document that is not referenced in the service's WSDL document is a three step process:

1. Convert the schema document to a WSDL document using the **`xsd2wsdl`** tool.
2. Generate Java for the types using the **`wsdl2java`** tool on the generated WSDL document.

IMPORTANT

You will get a warning from the **`wsdl2java`** tool stating that the WSDL document does not define any services. You can ignore this warning.

3. Add the generated classes to your classpath.

31.2. XML NAMESPACE MAPPING

Overview

XML Schema type, group, and element definitions are scoped using namespaces. The namespaces prevent possible naming clashes between entities that use the same name. Java packages serve a similar purpose. Therefore, Apache CXF maps the target namespace of a schema document into a package containing the classes necessary to implement the structures defined in the schema document.

Package naming

The name of the generated package is derived from a schema's target namespace using the following algorithm:

1. The URI scheme, if present, is stripped.

   NOTE

   Apache CXF will only strip the http:, https:, and urn: schemes.

   For example, the namespace **`http:\\www.widgetvendor.com\types\widgetTypes.xsd`** becomes **`\\widgetvendor.com\types\widgetTypes.xsd`**.

2. The trailing file type identifier, if present, is stripped.

   For example, **`\\www.widgetvendor.com\types\widgetTypes.xsd`** becomes **`\\widgetvendor.com\types\widgetTypes`**.

3. The resulting string is broken into a list of strings using `/` and `:` as separators.

   So, **`\\www.widgetvendor.com\types\widgetTypes`** becomes the list **`{"www.widegetvendor.com", "types", "widgetTypes"}`**.

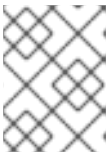
4. If the first string in the list is an internet domain name, it is decomposed as follows:

   a. The leading `www.` is stripped.

   b. The remaining string is split into its component parts using the `.` as the separator.

   c. The order of the list is reversed.

   So, **`{"www.widegetvendor.com", "types", "widgetTypes"}`** becomes **`{"com", "widegetvendor", "types", "widgetTypes"}`**

   NOTE

   Internet domain names end in one of the following: **`.com`**, **`.net`**, **`.edu`**, **`.org`**, **`.gov`**, or in one of the two-letter country codes.

5. The strings are converted into all lower case.

   So, **`{"com", "widegetvendor", "types", "widgetTypes"}`** becomes **`{"com", "widegetvendor", "types", "widgettypes"}`**.

6. The strings are normalized into valid Java package name components as follows:

   a. If the strings contain any special characters, the special characters are converted to an underscore(_).

   b. If any of the strings are a Java keyword, the keyword is prefixed with an underscore(_).

   c. If any of the strings begin with a numeral, the string is prefixed with an underscore(_).

7. The strings are concatenated using `.` as a separator.

   So, **`{"com", "widegetvendor", "types", "widgettypes"}`** becomes the package name com.widgetvendor.types.widgettypes.

The XML Schema constructs defined in the namespace **`http:\\www.widgetvendor.com\types\widgetTypes.xsd`** are mapped to the Java package com.widgetvendor.types.widgettypes.

Package contents

A JAXB generated package contains the following:

- A class implementing each complex type defined in the schema

  For more information on complex type mapping see Chapter 34, *Using Complex Types*.

- An enum type for any simple types defined using the **`enumeration`** facet

  For more information on how enumerations are mapped see Section 33.3, "Enumerations".

- A public **`ObjectFactory`** class that contains methods for instantiating objects from the schema

  For more information on the **`ObjectFactory`** class see Section 31.3, "The Object Factory".

- A **`package-info.java`** file that provides metadata about the classes in the package

31.3. THE OBJECT FACTORY

Overview

JAXB uses an object factory to provide a mechanism for instantiating instances of JAXB generated constructs. The object factory contains methods for instantiating all of the XML schema defined constructs in the package's scope. The only exception is that enumerations do not get a creation method in the object factory.

Complex type factory methods

For each Java class generated to implement an XML schema complex type, the object factory contains a method for creating an instance of the class. This method takes the form:

```
typeName createtypeName();
```

For example, if your schema contained a complex type named **`widgetType`**, Apache CXF generates a class called **`WidgetType`** to implement it. Example 31.5, "Complex Type Object Factory Entry" shows the generated creation method in the object factory.

Example 31.5. Complex Type Object Factory Entry

```
public class ObjectFactory
{
  ...
  WidgetType createWidgetType()
  {
    return new WidgetType();
```

```
    }
    ...
}
```

Element factory methods

For elements that are declared in the schema's global scope, Apache CXF inserts a factory method into the object factory. As discussed in Chapter 32, *Using XML Elements*, XML Schema elements are mapped to **JAXBElement<T>** objects. The creation method takes the form:

```
public JAXBElement<elementType> createelementName(elementType value);
```

For example if you have an element named **comment** of type xsd:string, Apache CXF generates the object factory method shown in Example 31.6, "Element Object Factory Entry"

Example 31.6. Element Object Factory Entry

```
public class ObjectFactory
{
  ...
    @XmlElementDecl(namespace = "...", name = "comment")
    public JAXBElement<String> createComment(String value) {
        return new JAXBElement<String>(_Comment_QNAME, String.class,
null, value);
    }
  ...
}
```

31.4. ADDING CLASSES TO THE RUNTIME MARSHALLER

Overview

When the Apache CXF runtime reads and writes XML data it uses a map that associates the XML Schema types with their representative Java types. By default, the map contains all of the types defined in the target namespace of the WSDL contract's **schema** element. It also contains any types that are generated from the namespaces of any schemas that are imported into the WSDL contract.

The addition of types from namespaces other than the schema namespace used by an application's **schema** element is accomplished using the **@XmlSeeAlso** annotation. If your application needs to work with types that are generated outside the scope of your application's WSDL document, you can edit the **@XmlSeeAlso** annotation to add them to the JAXB map.

Using the @XmlSeeAlso annotation

The **@XmlSeeAlso** annotation can be added to the SEI of your service. It contains a comma separated list of classes to include in the JAXB context. Example 31.7, "Syntax for Adding Classes to the JAXB Context" shows the syntax for using the **@XmlSeeAlso** annotation.

Example 31.7. Syntax for Adding Classes to the JAXB Context

```
import javax.xml.bind.annotation.XmlSeeAlso;
@WebService()
@XmlSeeAlso({Class1.class, Class2.class, ..., ClassN.class})
   public class GeneratedSEI {
       ...
    }
```

TIP

In cases where you have access to the JAXB generated classes, it is more efficient to use the **ObjectFactory** classes generated to support the needed types. Including the **ObjectFactory** class includes all of the classes that are known to the object factory.

Example

Example 31.8, "Adding Classes to the JAXB Context" shows an SEI annotated with **@XmlSeeAlso**.

Example 31.8. Adding Classes to the JAXB Context

```
...
import javax.xml.bind.annotation.XmlSeeAlso;
...
@WebService()
@XmlSeeAlso({org.apache.schemas.types.test.ObjectFactory.class,org.apach
e.schemas.tests.group_test.ObjectFactory.class})
   public interface Foo {
       ...
   }
```

CHAPTER 32. USING XML ELEMENTS

Abstract

XML Schema elements are used to define an instance of an element in an XML document. Elements are defined either in the global scope of an XML Schema document, or they are defined as a member of a complex type. When they are defined in the global scope, Apache CXF maps them to a JAXB element class that makes manipulating them easier.

OVERVIEW

An element instance in an XML document is defined by an XML Schema **`element`** element in the global scope of an XML Schema document To make it easier for Java developers to work with elements, Apache CXF maps globally scoped elements to either a special JAXB element class or to a Java class that is generated to match its content type.

How the element is mapped depends on if the element is defined using a named type referenced by the **`type`** attribute or if the element is defined using an in-line type definition. Elements defined with in-line type definitions are mapped to Java classes.

TIP

It is recommended that elements are defined using a named type because in-line types are not reusable by other elements in the schema.

XML SCHEMA MAPPING

In XML Schema elements are defined using **`element`** elements. **`element`** elements has one required attribute. The **`name`** specifies the name of the element as it appears in an XML document.

In addition to the **`name`** attribute **`element`** elements have the optional attributes listed in Table 32.1, “Attributes Used to Define an Element”.

Table 32.1. Attributes Used to Define an Element

Attribute	Description
`type`	Specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. If this attribute is not specified, you will need to include an in-line type definition.
`nillable`	Specifies if an element can be left out of a document entirely. If **`nillable`** is set to **`true`**, the element can be omitted from any document generated using the schema.

Attribute	Description
abstract	Specifies if an element can be used in an instance document. **true** indicates that the element cannot appear in the instance document. Instead, another element whose **substitutionGroup** attribute contains the QName of this element must appear in this element's place. For information on how this attribute effects code generation see the section called "Java mapping of abstract elements".
substitutionGroup	Specifies the name of an element that can be substituted with this element. For more information on using type substitution see Chapter 36, *Element Substitution*.
default	Specifies a default value for an element. For information on how this attribute effects code generation see the section called "Java mapping of elements with a default value".
fixed	Specifies a fixed value for the element.

Example 32.1, "Simple XML Schema Element Definition" shows a simple element definition.

Example 32.1. Simple XML Schema Element Definition

```
<element name="joeFred" type="xsd:string" />
```

An element can also define its own type using an in-line type definition. In-line types are specified using either a **complexType** element or a **simpleType** element. Once you specify whether the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data.

Example 32.2, "XML Schema Element Definition with an In-Line Type" shows an element definition with an in-line type definition.

Example 32.2. XML Schema Element Definition with an In-Line Type

```
<element name="skate">
  <complexType>
    <sequence>
      <element name="numWheels" type="xsd:int" />
      <element name="brand" type="xsd:string" />
    </sequence>
  </complexType>
</element>
```

JAVA MAPPING OF ELEMENTS WITH A NAMED TYPE

By default, globally defined elements are mapped to **JAXBElement<T>** objects where the template class is determined by the value of the **element** element's **type** attribute. For primitive types, the template class is derived using the wrapper class mapping described in the section called "Wrapper classes". For complex types, the Java class generated to support the complex type is used as the template class.

To support the mapping and to relieve the developer of unnecessary worry about an element's QName, an object factory method is generated for each globally defined element, as shown in Example 32.3, "Object Factory Method for a Globally Scoped Element".

Example 32.3. Object Factory Method for a Globally Scoped Element

```
public class ObjectFactory {

    private final static QName _name_QNAME = new
QName("targetNamespace", "localName");

  ...

   @XmlElementDecl(namespace = "targetNamespace", name = "localName")
    public JAXBElement<type> createname(type value);

}
```

For example, the element defined in Example 32.1, "Simple XML Schema Element Definition" results in the object factory method shown in Example 32.4, "Object Factory for a Simple Element" .

Example 32.4. Object Factory for a Simple Element

```
public class ObjectFactory {

    private final static QName _JoeFred_QNAME = new QName("...",
"joeFred");

  ...

   @XmlElementDecl(namespace = "...", name = "joeFred")
    public JAXBElement<String> createJoeFred(String value);

}
```

Example 32.5, "Using a Globally Scoped Element" shows an example of using a globally scoped element in Java.

Example 32.5. Using a Globally Scoped Element

```
JAXBElement<String> element = createJoeFred("Green");
String color = element.getValue();
```

USING ELEMENTS WITH NAMED TYPES IN WSDL

If a globally scoped element is used to define a message part, the generated Java parameter is not an instance of **JAXBElement<T>**. Instead it is mapped to a regular Java type or class.

Given the WSDL fragment shown in Example 32.6, "WSDL Using an Element as a Message Part" , the resulting method has a parameter of type String.

Example 32.6. WSDL Using an Element as a Message Part

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"

targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"

xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema
targetNamespace="http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"><element name="sayHi">
      <element name="sayHi" type="string"/>
      <element name="sayHiResponse" type="string"/>
    </schema>
  </wsdl:types>

  <wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
  </wsdl:message>
  <wsdl:message name="sayHiResponse">
    <wsdl:part element="x1:sayHiResponse" name="out"/>
  </wsdl:message>

  <wsdl:portType name="Greeter">
    <wsdl:operation name="sayHi">
      <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
      <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

Example 32.7, "Java Method Using a Global Element as a Part" shows the generated method signature for the **sayHi** operation.

Example 32.7. Java Method Using a Global Element as a Part

```
String sayHi(String in);
```

JAVA MAPPING OF ELEMENTS WITH AN IN-LINE TYPE

When an element is defined using an in-line type, it is mapped to Java following the same rules used for mapping other types to Java. The rules for simple types are described in Chapter 33, *Using Simple Types*. The rules for complex types are described in Chapter 34, *Using Complex Types*.

When a Java class is generated for an element with an in-line type definition, the generated class is decorated with the **@XmlRootElement** annotation. The **@XmlRootElement** annotation has two useful properties: name and namespace. These attributes are described in Table 32.2, "Properties for the @XmlRootElement Annotation".

Table 32.2. Properties for the @XmlRootElement Annotation

Property	Description
name	Specifies the value of the XML Schema **element** element's **name** attribute.
namespace	Specifies the namespace in which the element is defined. If this element is defined in the target namespace, the property is not specified.

The **@XmlRootElement** annotation is not used if the element meets one or more of the following conditions:

- The element's **nillable** attribute is set to **true**
- The element is the head element of a substitution group

  For more information on substitution groups see Chapter 36, *Element Substitution*.

JAVA MAPPING OF ABSTRACT ELEMENTS

When the element's **abstract** attribute is set to **true** the object factory method for instantiating instances of the type is not generated. If the element is defined using an in-line type, the Java class supporting the in-line type is generated.

JAVA MAPPING OF ELEMENTS WITH A DEFAULT VALUE

When the element's **default** attribute is used the defaultValue property is added to the generated **@XmlElementDecl** annotation. For example, the element defined in Example 32.8, "XML Schema Element with a Default Value" results in the object factory method shown in Example 32.9, "Object Factory Method for an Element with a Default Value".

Example 32.8. XML Schema Element with a Default Value

```
<element name="size" type="xsd:int" default="7"/>
```

Example 32.9. Object Factory Method for an Element with a Default Value

```
@XmlElementDecl(namespace = "...", name = "size", defaultValue = "7")
```

```
    public JAXBElement<Integer> createUnionJoe(Integer value) {
        return new JAXBElement<Integer>(_Size_QNAME, Integer.class,
null, value);
    }
```

CHAPTER 33. USING SIMPLE TYPES

Abstract

XML Schema simple types are either XML Schema primitive types like xsd:int, or are defined using the `simpleType` element. They are used to specify elements that do not contain any children or attributes. They are generally mapped to native Java constructs and do not require the generation of special classes to implement them. Enumerated simple types do not result in generated code because they are mapped to Java enum types.

33.1. PRIMITIVE TYPES

Overview

When a message part is defined using one of the XML Schema primitive types, the generated parameter's type is mapped to a corresponding Java native type. The same pattern is used when mapping elements that are defined within the scope of a complex type. The resulting field is of the corresponding Java native type.

Mappings

Table 33.1, “XML Schema Primitive Type to Java Native Type Mapping” lists the mapping between XML Schema primitive types and Java native types.

Table 33.1. XML Schema Primitive Type to Java Native Type Mapping

XML Schema Type	Java Type
xsd:string	`String`
xsd:integer	`BigInteger`
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	`BigDecimal`
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte

XML Schema Type	Java Type
xsd:QName	**`QName`**
xsd:dateTime	**`XMLGregorianCalendar`**
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	**`XMLGregorianCalendar`**
xsd:date	**`XMLGregorianCalendar`**
xsd:g	**`XMLGregorianCalendar`**
xsd:anySimpleType [a]	**`Object`**
xsd:anySimpleType [b]	**`String`**
xsd:duration	**`Duration`**
xsd:NOTATION	**`QName`**

[a] For elements of this type.

[b] For attributes of this type.

Wrapper classes

Mapping XML Schema primitive types to Java primitive types does not work for all possible XML Schema constructs. Several cases require that an XML Schema primitive type is mapped to the Java primitive type's corresponding wrapper type. These cases include:

- An **`element`** element with its **`nillable`** attribute set to **`true`** as shown:

```
<element name="finned" type="xsd:boolean"
         nillable="true" />
```

- An **`element`** element with its **`minOccurs`** attribute set to **`0`** and its **`maxOccurs`** attribute set to **`1`**, or its **`maxOccurs`** attribute not specified, as shown :

```
<element name="plane" type="xsd:string" minOccurs="0" />
```

- An **attribute** element with its **use** attribute set to **optional**, or not specified, and having neither its **default** attribute nor its **fixed** attribute specified, as shown:

```
<element name="date">
  <complexType>
    <sequence/>
    <attribute name="calType" type="xsd:string"
               use="optional" />
  </complexType>
</element>
```

Table 33.2, "Primitive Schema Type to Java Wrapper Class Mapping" shows how XML Schema primitive types are mapped into Java wrapper classes in these cases.

Table 33.2. Primitive Schema Type to Java Wrapper Class Mapping

Schema Type	Java Type
xsd:int	**java.lang.Integer**
xsd:long	**java.lang.Long**
xsd:short	**java.lang.Short**
xsd:float	**java.lang.Float**
xsd:double	**java.lang.Double**
xsd:boolean	**java.lang.Boolean**
xsd:byte	**java.lang.Byte**
xsd:unsignedByte	**java.lang.Short**
xsd:unsignedShort	**java.lang.Integer**
xsd:unsignedInt	**java.lang.Long**
xsd:unsignedLong	**java.math.BigInteger**
xsd:duration	**java.lang.String**

33.2. SIMPLE TYPES DEFINED BY RESTRICTION

Overview

XML Schema allows you to create simple types by deriving a new type from another primitive type or simple type. Simple types are described using a **simpleType** element.

The new types are described by restricting the *base type* with one or more facets. These facets limit the possible valid values that can be stored in the new type. For example, you could define a simple type, SSN, which is a string of exactly 9 characters.

Each of the primitive XML Schema types has their own set of optional facets.

Procedure

To define your own simple type do the following:

1. Determine the base type for your new simple type.
2. Determine what restrictions define the new type based on the available facets for the chosen base type.
3. Using the syntax shown in this section, enter the appropriate **simpleType** element into the types section of your contract.

Defining a simple type in XML Schema

Example 33.1, "Simple type syntax" shows the syntax for describing a simple type.

Example 33.1. Simple type syntax

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value" />
    <facet value="value" />
    ...
  </restriction>
</simpleType>
```

The type description is enclosed in a **simpleType** element and identified by the value of the **name** attribute. The base type from which the new simple type is being defined is specified by the **base** attribute of the **xsd:restriction** element. Each facet element is specified within the **restriction** element. The available facets and their valid settings depend on the base type. For example, xsd:string has a number of facets including:

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whitespace**

Example 33.2, "Postal Code Simple Type " shows the definition for a simple type that represents the two-letter postal code used for US states. It can only contain two, uppercase letters. **TX** is a valid value, but **tx** or **tX** are not valid values.

Example 33.2. Postal Code Simple Type

```
<xsd:simpleType name="postalCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{2}" />
  </xsd:restriction>
</xsd:simpleType>
```

Mapping to Java

Apache CXF maps user-defined simple types to the Java type of the simple type's base type. So, any message using the simple type postalCode, shown in Example 33.2, "Postal Code Simple Type " , is mapped to a **`String`** because the base type of postalCode is xsd:string. For example, the WSDL fragment shown in Example 33.3, "Credit Request with Simple Types" results in a Java method, **`state()`**, that takes a parameter, ***`postalCode`***, of **`String`**.

Example 33.3. Credit Request with Simple Types

```
<message name="stateRequest">
  <part name="postalCode" type="postalCode" />
</message>
...
<portType name="postalSupport">
  <operation name="state">
    <input message="tns:stateRequest" name="stateRec" />
    <output message="tns:stateResponse" name="credResp" />
  </operation>
</portType>
```

Enforcing facets

By default, Apache CXF does not enforce any of the facets that are used to restrict a simple type. However, you can configure Apache CXF endpoints to enforce the facets by enabling schema validation.

To configure Apache CXF endpoints to use schema validation set the schema-validation-enabled property to **`true`**. Example 33.4, "Service Provider Configured to Use Schema Validation" shows the configuration for a service provider that uses schema validation

Example 33.4. Service Provider Configured to Use Schema Validation

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                wsdlLocation="wsdl/hello_world.wsdl"
                createdFromAPI="true">
  <jaxws:properties>
    <entry key="schema-validation-enabled" value="true" />
  </jaxws:properties>
</jaxws:endpoint>
```

For more information on configuring Apache CXF see Part IV, “Configuring Web Service Endpoints” .

33.3. ENUMERATIONS

Overview

In XML Schema, enumerated types are simple types that are defined using the **xsd:enumeration** facet. Unlike atomic simple types, they are mapped to Java enums.

Defining an enumerated type in XML Schema

Enumerations are a simple type using the **xsd:enumeration** facet. Each **xsd:enumeration** facet defines one possible value for the enumerated type.

Example 33.5, “XML Schema Defined Enumeration” shows the definition for an enumerated type. It has the following possible values:

- **big**
- **large**
- **mungo**
- **gargantuan**

Example 33.5. XML Schema Defined Enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
    <enumeration value="gargantuan"/>
  </restriction>
```

Mapping to Java

XML Schema enumerations where the base type is xsd:string are automatically mapped to Java enum type. You can instruct the code generator to map enumerations with other base types to Java enum types by using the customizations described in Section 37.4, “Customizing Enumeration Mapping” .

The enum type is created as follows:

1. The name of the type is taken from the **name** attribute of the simple type definition and converted to a Java identifier.

   In general, this means converting the first character of the XML Schema's name to an uppercase letter. If the first character of the XML Schema's name is an invalid character, an underscrore (_) is prepended to the name.

2. For each **enumeration** facet, an enum constant is generated based on the value of the **value** attribute.

The constant's name is derived by converting all of the lowercase letters in the value to their uppercase equivalent.

3. A constructor is generated that takes the Java type mapped from the enumeration's base type.
4. A public method called **`value()`** is generated to access the facet value that is represented by an instance of the type.

   The return type of the **`value()`** method is the base type of the XML Schema type.
5. A public method called **`fromValue()`** is generated to create an instance of the enum type based on a facet value.

   The parameter type of the **`value()`** method is the base type of the XML Schema type.
6. The class is decorated with the **`@XmlEnum`** annotation.

The enumerated type defined in Example 33.5, "XML Schema Defined Enumeration" is mapped to the enum type shown in Example 33.6, "Generated Enumerated Type for a String Bases XML Schema Enumeration".

Example 33.6. Generated Enumerated Type for a String Bases XML Schema Enumeration

```
@XmlType(name = "widgetSize")
@XmlEnum
public enum WidgetSize {

    @XmlEnumValue("big")
    BIG("big"),
    @XmlEnumValue("large")
    LARGE("large"),
    @XmlEnumValue("mungo")
    MUNGO("mungo"),
    @XmlEnumValue("gargantuan")
    GARGANTUAN("gargantuan");
    private final String value;

    WidgetSize(String v) {
        value = v;
    }

    public String value() {
        return value;
    }

    public static WidgetSize fromValue(String v) {
        for (WidgetSize c: WidgetSize.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }

}
```

33.4. LISTS

Overview

XML Schema supports a mechanism for defining data types that are a list of space separated simple types. An example of an element, **primeList**, using a list type is shown in Example 33.7, "List Type Example".

Example 33.7. List Type Example

```
<primeList>1 3 5 7 9 11 13<\primeList>
```

XML Schema list types are generally mapped to Java **List<T>** objects. The only variation to this pattern is when a message part is mapped directly to an instance of an XML Schema list type.

Defining list types in XML Schema

XML Schema list types are simple types and as such are defined using a **simpleType** element. The most common syntax used to define a list type is shown in Example 33.8, "Syntax for XML Schema List Types".

Example 33.8. Syntax for XML Schema List Types

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value" />
    <facet value="value" />
    ...
  </list>
</simpleType>
```

The value given for *atomicType* defines the type of the elements in the list. It can only be one of the built in XML Schema atomic types, like xsd:int or xsd:string, or a user-defined simple type that is not a list.

In addition to defining the type of elements listed in the list type, you can also use facets to further constrain the properties of the list type. Table 33.3, "List Type Facets" shows the facets used by list types.

Table 33.3. List Type Facets

Facet	Effect
length	Defines the number of elements in an instance of the list type.

Facet	Effect
minLength	Defines the minimum number of elements allowed in an instance of the list type.
maxLength	Defines the maximum number of elements allowed in an instance of the list type.
enumeration	Defines the allowable values for elements in an instance of the list type.
pattern	Defines the lexical form of the elements in an instance of the list type. Patterns are defined using regular expressions.

For example, the definition for the **simpleList** element shown in Example 33.7, "List Type Example", is shown in Example 33.9, "Definition of a List Type".

Example 33.9. Definition of a List Type

```
<simpleType name="primeListType">
  <list itemType="int"/>
</simpleType>
<element name="primeList" type="primeListType"/>
```

In addition to the syntax shown in Example 33.8, "Syntax for XML Schema List Types" you can also define a list type using the less common syntax shown in Example 33.10, "Alternate Syntax for List Types".

Example 33.10. Alternate Syntax for List Types

```
<simpleType name="listType">
  <list>
    <simpleType>
      <restriction base="atomicType">
        <facet value="value"/>
        <facet value="value"/>
        ...
      </restriction>
    </simpleType>
  </list>
  </simpleType>
```

Mapping list type elements to Java

When an element is defined a list type, the list type is mapped to a collection property. A collection property is a Java **List<T>** object. The template class used by the **List<T>** is the wrapper class mapped from the list's base type. For example, the list type defined in Example 33.9, "Definition of a List Type" is mapped to a **List<Integer>**.

For more information on wrapper type mapping see the section called “Wrapper classes”.

Mapping list type parameters to Java

When a message part is defined as a list type, or is mapped to an element of a list type, the resulting method parameter is mapped to an array instead of a **List<T>** object. The base type of the array is the wrapper class of the list type's base class.

For example, the WSDL fragment in Example 33.11, “WSDL with a List Type Message Part” results in the method signature shown in Example 33.12, “Java Method with a List Type Parameter” .

Example 33.11. WSDL with a List Type Message Part

```
<definitions ...>
  ...
  <types ...>
    <schema ... >
      <simpleType name="primeListType">
        <list itemType="int"/>
      </simpleType>
      <element name="primeList" type="primeListType"/>
    </schemas>
  </types>
  <message name="numRequest">
    <part name="inputData" element="xsd1:primeList" />
  </message>
  <message name="numResponse">;
    <part name="outputData" type="xsd:int">
  ...
  <portType name="numberService">
    <operation name="primeProcessor">
      <input name="numRequest" message="tns:numRequest" />
      <output name="numResponse" message="tns:numResponse" />
    </operation>
    ...
  </portType>
  ...
</definitions>
```

Example 33.12. Java Method with a List Type Parameter

```
public interface NumberService {

    @XmlList
    @WebResult(name = "outputData", targetNamespace = "", partName =
"outputData")
    @WebMethod
    public int primeProcessor(
        @WebParam(partName = "inputData", name = "primeList",
targetNamespace = "...")
        java.lang.Integer[] inputData
    );
}
```

33.5. UNIONS

Overview

In XML Schema, a union is a construct that allows you to describe a type whose data can be one of a number of simple types. For example, you can define a type whose value is either the integer **1** or the string **first**. Unions are mapped to Java **Strings**.

Defining in XML Schema

XML Schema unions are defined using a **simpleType** element. They contain at least one **union** element that defines the member types of the union. The member types of the union are the valid types of data that can be stored in an instance of the union. They are defined using the **union** element's **memberTypes** attribute. The value of the **memberTypes** attribute contains a list of one or more defined simple type names. Example 33.13, “Simple Union Type” shows the definition of a union that can store either an integer or a string.

Example 33.13. Simple Union Type

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

In addition to specifying named types as a member type of a union, you can also define an anonymous simple type as a member type of a union. This is done by adding the anonymous type definition inside of the **union** element. Example 33.14, “Union with an Anonymous Member Type” shows an example of a union containing an anonymous member type that restricts the possible values of a valid integer to the range 1 through 10.

Example 33.14. Union with an Anonymous Member Type

```
<simpleType name="restrictedOrderNumUnion">
  <union memberTypes="xsd:string">
    <simpleType>
      <restriction base="xsd:int">
        <minInclusive value="1" />
        <maxInclusive value="10" />
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

Mapping to Java

XML Schema union types are mapped to Java **String** objects. By default, Apache CXF does not validate the contents of the generated object. To have Apache CXF validate the contents you will must configure the runtime to use schema validation as described in the section called “Enforcing facets”.

33.6. SIMPLE TYPE SUBSTITUTION

Overview

XML allows for simple type substitution between compatible types using the **xsi:type** attribute. The default mapping of simple types to Java primitive types, however, does not fully support simple type substitution. The runtime can handle basic simple type substitution, but information is lost. The code generators can be customized to generate Java classes that facilitate lossless simple type substitution.

Default mapping and marshaling

Because Java primitive types do not support type substitution, the default mapping of simple types to Java primitive types presents problems for supporting simple type substitution. The Java virtual machine will balk if an attempt is made to pass a short into a variable that expects an int even though the schema defining the types allows it.

To get around the limitations imposed by the Java type system, Apache CXF allows for simple type substitution when the value of the element's **xsi:type** attribute meets one of the following conditions:

- It specifies a primitive type that is compatible with the element's schema type.
- It specifies a type that derives by restriction from the element's schema type.
- It specifies a complex type that derives by extension from the element's schema type.

When the runtime does the type substitution it does not retain any knowledge of the type specified in the element's **xsi:type** attribute. If the type substitution is from a complex type to a simple type, only the value directly related to the simple type is preserved. Any other elements and attributes added by extension are lost.

Supporting lossless type substitution

You can customize the generation of simple types to facilitate lossless support of simple type substitution in the following ways:

- Set the **globalBindings** customization element's **mapSimpleTypeDef** to **true**.

  This instructs the code generator to create Java value classes for all named simple types defined in the global scope.

  For more information see Section 37.3, "Generating Java Classes for Simple Types" .

- Add a **javaType** element to the **globalBindings** customization element.

  This instructs the code generators to map all instances of an XML Schema primitive type to s specific class of object.

  For more information see Section 37.2, "Specifying the Java Class of an XML Schema Primitive".

- Add a **baseType** customization element to the specific elements you want to customize.

The **baseType** customization element allows you to specify the Java type generated to represent a property. To ensure the best compatibility for simple type substitution, use `java.lang.Object` as the base type.

For more information see Section 37.6, “Specifying the Base Type of an Element or an Attribute”.

CHAPTER 34. USING COMPLEX TYPES

Abstract

Complex types can contain multiple elements and they can have attributes. They are mapped into Java classes that can hold the data represented by the type definition. Typically, the mapping is to a bean with a set of properties representing the elements and the attributes of the content model..

34.1. BASIC COMPLEX TYPE MAPPING

Overview

XML Schema complex types define constructs containing more complex information than a simple type. The most simple complex types define an empty element with an attribute. More intricate complex types are made up of a collection of elements.

By default, an XML Schema complex type is mapped to a Java class, with a member variable to represent each element and attribute listed in the XML Schema definition. The class has setters and getters for each member variable.

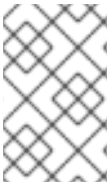
Defining in XML Schema

XML Schema complex types are defined using the `complexType` element. The `complexType` element wraps the rest of elements used to define the structure of the data. It can appear either as the parent element of a named type definition, or as the child of an `element` element anonymously defining the structure of the information stored in the element. When the `complexType` element is used to define a named type, it requires the use of the `name` attribute. The `name` attribute specifies a unique identifier for referencing the type.

Complex type definitions that contain one or more elements have one of the child elements described in Table 34.1, “Elements for Defining How Elements Appear in a Complex Type” . These elements determine how the specified elements appear in an instance of the type.

Table 34.1. Elements for Defining How Elements Appear in a Complex Type

Element	Description
`all`	All of the elements defined as part of the complex type must appear in an instance of the type. However, they can appear in any order.
`choice`	Only one of the elements defined as part of the complex type can appear in an instance of the type.
`sequence`	All of the elements defined as part of the complex type must appear in an instance of the type, and they must also appear in the order specified in the type definition.



NOTE

If a complex type definition only uses attributes, you do not need one of the elements described in Table 34.1, "Elements for Defining How Elements Appear in a Complex Type".

After deciding how the elements will appear, you define the elements by adding one or more **element** element children to the definition.

Example 34.1, "XML Schema Complex Type" shows a complex type definition in XML Schema.

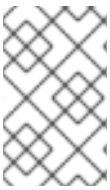
Example 34.1. XML Schema Complex Type

```
<complexType name="sequence">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="street" type="xsd:short" />
    <element name="city" type="xsd:string" />
    <element name="state" type="xsd:string" />
    <element name="zipCode" type="xsd:string" />
  </sequence>
</complexType>
```

Mapping to Java

XML Schema complex types are mapped to Java classes. Each element in the complex type definition is mapped to a member variable in the Java class. Getter and setter methods are also generated for each element in the complex type.

All generated Java classes are decorated with the **@XmlType** annotation. If the mapping is for a named complex type, the annotations name is set to the value of the **complexType** element's **name** attribute. If the complex type is defined as part of an element definition, the value of the **@XmlType** annotation's name property is the value of the **element** element's **name** attribute.



NOTE

As described in the section called "Java mapping of elements with an in-line type" , the generated class is decorated with the **@XmlRootElement** annotation if it is generated for a complex type defined as part of an element definition.

To provide the runtime with guidelines indicating how the elements of the XML Schema complex type should be handled, the code generators alter the annotations used to decorate the class and its member variables.

All Complex Type

All complex types are defined using the **all** element. They are annotated as follows:

- The **@XmlType** annotation's propOrder property is empty.
- Each element is decorated with the **@XmlElement** annotation.
- The **@XmlElement** annotation's required property is set to **true**.

Example 34.2, "Mapping of an All Complex Type" shows the mapping for an all complex type with two elements.

Example 34.2. Mapping of an All Complex Type

```
@XmlType(name = "all", propOrder = {

})
public class All {
    @XmlElement(required = true)
    protected BigDecimal amount;
    @XmlElement(required = true)
    protected String type;

    public BigDecimal getAmount() {
        return amount;
    }

    public void setAmount(BigDecimal value) {
        this.amount = value;
    }

    public String getType() {
        return type;
    }

    public void setType(String value) {
        this.type = value;
    }
}
```

Choice Complex Type

Choice complex types are defined using the **choice** element. They are annotated as follows:

- The **@XmlType** annotation's propOrder property lists the names of the elements in the order they appear in the XML Schema definition.
- None of the member variables are annotated.

Example 34.3, "Mapping of a Choice Complex Type" shows the mapping for a choice complex type with two elements.

Example 34.3. Mapping of a Choice Complex Type

```
@XmlType(name = "choice", propOrder = {
    "address",
    "floater"
})
public class Choice {

    protected Sequence address;
    protected Float floater;
```

```
    public Sequence getAddress() {
        return address;
    }

    public void setAddress(Sequence value) {
        this.address = value;
    }

    public Float getFloater() {
        return floater;
    }

    public void setFloater(Float value) {
        this.floater = value;
    }

}
```

Sequence Complex Type

A sequence complex type is defined using the **sequence** element. It is annotated as follows:

- The **@XmlType** annotation's propOrder property lists the names of the elements in the order they appear in the XML Schema definition.
- Each element is decorated with the **@XmlElement** annotation.
- The **@XmlElement** annotation's required property is set to **true**.

Example 34.4, "Mapping of a Sequence Complex Type" shows the mapping for the complex type defined in Example 34.1, "XML Schema Complex Type".

Example 34.4. Mapping of a Sequence Complex Type

```
@XmlType(name = "sequence", propOrder = {
    "name",
    "street",
    "city",
    "state",
    "zipCode"
})
public class Sequence {

    @XmlElement(required = true)
    protected String name;
    protected short street;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String state;
    @XmlElement(required = true)
    protected String zipCode;

    public String getName() {
        return name;
```

```
    }

    public void setName(String value) {
        this.name = value;
    }

    public short getStreet() {
        return street;
    }

    public void setStreet(short value) {
        this.street = value;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String value) {
        this.city = value;
    }

    public String getState() {
        return state;
    }

    public void setState(String value) {
        this.state = value;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String value) {
        this.zipCode = value;
    }
}
```

34.2. ATTRIBUTES

Overview

Apache CXF supports the use of **`attribute`** elements and **`attributeGroup`** elements within the scope of a **`complexType`** element. When defining structures for an XML document attribute declarations provide a means of adding information that is specified within the tag, not the value that the tag contains. For example, when describing the XML element <value currency="euro">410<\value> in XML Schema the **`currency`** attribute is described using an **`attribute`** element as shown in Example 34.5, “XML Schema Defining and Attribute” .

The **`attributeGroup`** element allows you to define a group of reusable attributes that can be referenced by all complex types defined by the schema. For example, if you are defining a series of

elements that all use the attributes **category** and **pubDate**, you could define an attribute group with these attributes and reference them in all the elements that use them. This is shown in Example 34.7, "Attribute Group Definition".

When describing data types for use in developing application logic, attributes whose **use** attribute is set to either **optional** or **required** are treated as elements of a structure. For each attribute declaration contained within a complex type description, an element is generated in the class for the attribute, along with the appropriate getter and setter methods.

Defining an attribute in XML Schema

An XML Schema **attribute** element has one required attribute, **name**, that is used to identify the attribute. It also has four optional attributes that are described in Table 34.2, "Optional Attributes Used to Define Attributes in XML Schema".

Table 34.2. Optional Attributes Used to Define Attributes in XML Schema

Attribute	Description
use	Specifies if the attribute is required. Valid values are **required**, **optional**, or **prohibited**. **optional** is the default value.
type	Specifies the type of value the attribute can take. If it is not used the schema type of the attribute must be defined in-line.
default	Specifies a default value to use for the attribute. It is only used when the **attribute** element's **use** attribute is set to **optional**.
fixed	Specifies a fixed value to use for the attribute. It is only used when the **attribute** element's **use** attribute is set to **optional**.

Example 34.5, "XML Schema Defining and Attribute" shows an attribute element defining an attribute, currency, whose value is a string.

Example 34.5. XML Schema Defining and Attribute

```
<element name="value">
  <complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="currency" type="xsd:string"
                       use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

If the **type** attribute is omitted from the **attribute** element, the format of the data must be described in-line. Example 34.6, "Attribute with an In-Line Data Description" shows an **attribute** element for an attribute, **category**, that can take the values **autobiography**, **non-fiction**, or **fiction**.

Example 34.6. Attribute with an In-Line Data Description

```
<attribute name="category" use="required">
  <simpleType>
    <restriction base="xsd:string">
      <enumeration value="autobiography"/>
      <enumeration value="non-fiction"/>
      <enumeration value="fiction"/>
    </restriction>
  </simpleType>
</attribute>
```

Using an attribute group in XML Schema

Using an attribute group in a complex type definition is a two step process:

1. Define the attribute group.

   An attribute group is defined using an **attributeGroup** element with a number of **attribute** child elements. The **attributeGroup** requires a **name** attribute that defines the string used to refer to the attribute group. The **attribute** elements define the members of the attribute group and are specified as shown in the section called "Defining an attribute in XML Schema". Example 34.7, "Attribute Group Definition" shows the description of the attribute group **catalogIndecies**. The attribute group has two members: **category**, which is optional, and **pubDate**, which is required.

   Example 34.7. Attribute Group Definition

   ```
   <attributeGroup name="catalogIndices">
     <attribute name="category" type="catagoryType" />
     <attribute name="pubDate" type="dateTime"
                use="required" />
   </attributeGroup>
   ```

2. Use the attribute group in the definition of a complex type.

   You use attribute groups in complex type definitions by using the **attributeGroup** element with the **ref** attribute. The value of the **ref** attribute is the name given the attribute group that you want to use as part of the type definition. For example if you want to use the attribute group **catalogIndecies** in the complex type dvdType, you would use <attributeGroup ref="catalogIndecies" /> as shown in Example 34.8, "Complex Type with an Attribute Group" .

   Example 34.8. Complex Type with an Attribute Group

   ```
   <complexType name="dvdType">
     <sequence>
   ```

```
      <element name="title" type="xsd:string" />
      <element name="director" type="xsd:string" />
      <element name="numCopies" type="xsd:int" />
    </sequence>
    <attributeGroup ref="catalogIndices" />
  </complexType>
```

Mapping attributes to Java

Attributes are mapped to Java in much the same way that member elements are mapped to Java. Required attributes and optional attributes are mapped to member variables in the generated Java class. The member variables are decorated with the **`@XmlAttribute`** annotation. If the attribute is required, the **`@XmlAttribute`** annotation's `required` property is set to **`true`**.

The complex type defined in Example 34.9, “techDoc Description ” is mapped to the Java class shown in Example 34.10, “techDoc Java Class”.

Example 34.9. techDoc Description

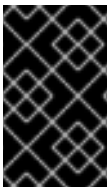
```
<complexType name="techDoc">
  <all>
    <element name="product" type="xsd:string" />
    <element name="version" type="xsd:short" />
  </all>
  <attribute name="usefullness" type="xsd:float"
             use="optional" default="0.01" />
</complexType>
```

Example 34.10. techDoc Java Class

```
@XmlType(name = "techDoc", propOrder = {

})
public class TechDoc {

    @XmlElement(required = true)
    protected String product;
    protected short version;
    @XmlAttribute
    protected Float usefullness;

    public String getProduct() {
        return product;
    }

    public void setProduct(String value) {
        this.product = value;
    }

    public short getVersion() {
        return version;
    }
```

```
    public void setVersion(short value) {
        this.version = value;
    }

    public float getUsefullness() {
        if (usefullness == null) {
            return 0.01F;
        } else {
            return usefullness;
        }
    }

    public void setUsefullness(Float value) {
        this.usefullness = value;
    }
}
```

As shown in Example 34.10, "techDoc Java Class", the **default** attribute and the **fixed** attribute instruct the code generators to add code to the getter method generated for the attribute. This additional code ensures that the specified value is returned if no value is set.



IMPORTANT

The **fixed** attribute is treated the same as the **default** attribute. If you want the **fixed** attribute to be treated as a Java constant you can use the customization described in Section 37.5, "Customizing Fixed Value Attribute Mapping".

Mapping attribute groups to Java

Attribute groups are mapped to Java as if the members of the group were explicitly used in the type definition. If the attribute group has three members, and it is used in a complex type, the generated class for that type will include a member variable, along with the getter and setter methods, for each member of the attribute group. For example, the complex type defined in Example 34.8, "Complex Type with an Attribute Group", Apache CXF generates a class containing the member variables **category** and **pubDate** to support the members of the attribute group as shown in Example 34.11, "dvdType Java Class".

Example 34.11. dvdType Java Class

```
@XmlType(name = "dvdType", propOrder = {
    "title",
    "director",
    "numCopies"
})
public class DvdType {

    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String director;
    protected int numCopies;
    @XmlAttribute
```

```
    protected CatagoryType category;
    @XmlAttribute(required = true)
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar pubDate;

    public String getTitle() {
        return title;
    }

    public void setTitle(String value) {
        this.title = value;
    }

    public String getDirector() {
        return director;
    }

    public void setDirector(String value) {
        this.director = value;
    }

    public int getNumCopies() {
        return numCopies;
    }

    public void setNumCopies(int value) {
        this.numCopies = value;
    }

    public CatagoryType getCatagory() {
        return catagory;
    }

    public void setCatagory(CatagoryType value) {
        this.catagory = value;
    }

    public XMLGregorianCalendar getPubDate() {
        return pubDate;
    }

    public void setPubDate(XMLGregorianCalendar value) {
        this.pubDate = value;
    }

}
```

34.3. DERIVING COMPLEX TYPES FROM SIMPLE TYPES

Overview

Apache CXF supports derivation of a complex type from a simple type. A simple type has, by definition, neither sub-elements nor attributes. Hence, one of the main reasons for deriving a complex type from a simple type is to add attributes to the simple type.

There are two ways of deriving a complex type from a simple type:

- By extension
- By restriction

Derivation by extension

Example 34.12, “Deriving a Complex Type from a Simple Type by Extension” shows an example of a complex type, internationalPrice, derived by extension from the xsd:decimal primitive type to include a currency attribute.

Example 34.12. Deriving a Complex Type from a Simple Type by Extension

```
<complexType name="internationalPrice">
    <simpleContent>
        <extension base="xsd:decimal">
            <attribute name="currency" type="xsd:string"/>
        </extension>
    </simpleContent>
    </complexType>
```

The **simpleContent** element indicates that the new type does not contain any sub-elements. The **extension** element specifies that the new type extends xsd:decimal.

Derivation by restriction

Example 34.13, “Deriving a Complex Type from a Simple Type by Restriction” shows an example of a complex type, idType, that is derived by restriction from xsd:string. The defined type restricts the possible values of xsd:stringto values that are ten characters in length. It also adds an attribute to the type.

Example 34.13. Deriving a Complex Type from a Simple Type by Restriction

```
<complexType name="idType">
  <simpleContent>
    <restriction base="xsd:string">
     <length value="10" />
     <attribute name="expires" type="xsd:dateTime" />
    </restriction>
  </simpleContent>
</complexType>
```

As in Example 34.12, “Deriving a Complex Type from a Simple Type by Extension” the **simpleContent** element signals that the new type does not contain any children. This example uses a **restriction** element to constrain the possible values used in the new type. The **attribute** element adds the element to the new type.

Mapping to Java

A complex type derived from a simple type is mapped to a Java class that is decorated with the

@XmlType annotation. The generated class contains a member variable, **value**, of the simple type from which the complex type is derived. The member variable is decorated with the **@XmlValue** annotation. The class also has a **getValue()** method and a **setValue()** method. In addition, the generated class has a member variable, and the associated getter and setter methods, for each attribute that extends the simple type.

Example 34.14, "idType Java Class" shows the Java class generated for the idType type defined in Example 34.13, "Deriving a Complex Type from a Simple Type by Restriction" .

Example 34.14. idType Java Class

```
@XmlType(name = "idType", propOrder = {
    "value"
})
public class IdType {

    @XmlValue
    protected String value;
    @XmlAttribute
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar expires;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public XMLGregorianCalendar getExpires() {
        return expires;
    }

    public void setExpires(XMLGregorianCalendar value) {
        this.expires = value;
    }

}
```

34.4. DERIVING COMPLEX TYPES FROM COMPLEX TYPES

Overview

Using XML Schema, you can derive new complex types by either extending or restricting other complex types using the **complexContent** element. When generating the Java class to represent the derived complex type, Apache CXF extends the base type's class. In this way, the generated Java code preserves the inheritance hierarchy intended in the XML Schema.

Schema syntax

You derive complex types from other complex types by using the **complexContent** element, and

either the **extension** element or the **restriction** element. The **complexContent** element specifies that the included data description includes more than one field. The **extension** element and the **restriction** element, which are children of the **complexContent** element, specify the base type being modified to create the new type. The base type is specified by the **base** attribute.

Extending a complex type

To extend a complex type use the **extension** element to define the additional elements and attributes that make up the new type. All elements that are allowed in a complex type description are allowable as part of the new type's definition. For example, you can add an anonymous enumeration to the new type, or you can use the **choice** element to specify that only one of the new fields can be valid at a time.

Example 34.15, "Deriving a Complex Type by Extension" shows an XML Schema fragment that defines two complex types, widgetOrderInfo and widgetOrderBillInfo. widgetOrderBillInfo is derived by extending widgetOrderInfo to include two new elements: **orderNumber** and **amtDue**.

Example 34.15. Deriving a Complex Type by Extension

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element name="order_date" type="xsd:dateTime"/>
    <element name="type" type="xsd1:widgetSize"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:boolean" use="optional" />
</complexType>
<complexType name="widgetOrderBillInfo">
  <complexContent>
    <extension base="xsd1:widgetOrderInfo">
      <sequence>
        <element name="amtDue" type="xsd:decimal"/>
        <element name="orderNumber" type="xsd:string"/>
      </sequence>
      <attribute name="paid" type="xsd:boolean"
                 default="false" />
    </extension>
  </complexContent>
</complexType>
```

Restricting a complex type

To restrict a complex type use the **restriction** element to limit the possible values of the base type's elements or attributes. When restricting a complex type you must list all of the elements and attributes of the base type. For each element you can add restrictive attributes to the definition. For example, you can add a **maxOccurs** attribute to an element to limit the number of times it can occur. You can also use the **fixed** attribute to force one or more of the elements to have predetermined values.

Example 34.16, "Defining a Complex Type by Restriction" shows an example of defining a complex type by restricting another complex type. The restricted type, wallawallaAddress, can only be used for addresses in Walla Walla, Washington because the values for the **city** element, the **state** element,

and the **zipCode** element are fixed.

Example 34.16. Defining a Complex Type by Restriction

```
<complexType name="Address">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="street" type="xsd:short" maxOccurs="3"/>
    <element name="city" type="xsd:string"/>
    <element name="state" type="xsd:string"/>
    <element name="zipCode" type="xsd:string"/>
  </sequence>
</complexType>
<complexType name="wallawallaAddress">
  <complexContent>
    <restriction base="xsd1:Address">
      <sequence>
        <element name="name" type="xsd:string"/>
        <element name="street" type="xsd:short"
                 maxOccurs="3"/>
        <element name="city" type="xsd:string"
                 fixed="WallaWalla"/>
        <element name="state" type="xsd:string"
                 fixed="WA" />
        <element name="zipCode" type="xsd:string"
                 fixed="99362" />
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

Mapping to Java

As it does with all complex types, Apache CXF generates a class to represent complex types derived from another complex type. The Java class generated for the derived complex type extends the Java class generated to support the base complex type. The base Java class is also modified to include the **@XmlSeeAlso** annotation. The base class' **@XmlSeeAlso** annotation lists all of the classes that extend the base class.

When the new complex type is derived by extension, the generated class will include member variables for all of the added elements and attributes. The new member variables will be generated according to the same mappings as all other elements.

When the new complex type is derived by restriction, the generated class will have no new member variables. The generated class will simply be a shell that does not provide any additional functionality. It is entirely up to you to ensure that the restrictions defined in the XML Schema are enforced.

For example, the schema in Example 34.15, “Deriving a Complex Type by Extension” results in the generation of two Java classes: **WidgetOrderInfo** and **WidgetBillOrderInfo**. **WidgetOrderBillInfo** extends **WidgetOrderInfo** because widgetOrderBillInfo is derived by extension from widgetOrderInfo. Example 34.17, “WidgetOrderBillInfo” shows the generated class for widgetOrderBillInfo.

Example 34.17. WidgetOrderBillInfo

```
@XmlType(name = "widgetOrderBillInfo", propOrder = {
    "amtDue",
    "orderNumber"
})
public class WidgetOrderBillInfo
    extends WidgetOrderInfo
{
    @XmlElement(required = true)
    protected BigDecimal amtDue;
    @XmlElement(required = true)
    protected String orderNumber;
    @XmlAttribute
    protected Boolean paid;

    public BigDecimal getAmtDue() {
        return amtDue;
    }

    public void setAmtDue(BigDecimal value) {
        this.amtDue = value;
    }

    public String getOrderNumber() {
        return orderNumber;
    }

    public void setOrderNumber(String value) {
        this.orderNumber = value;
    }

    public boolean isPaid() {
        if (paid == null) {
            return false;
        } else {
            return paid;
        }
    }

    public void setPaid(Boolean value) {
        this.paid = value;
    }
}
```

34.5. OCCURRENCE CONSTRAINTS

XML Schema allows you to specify the occurrence constraints on four of the XML Schema elements that make up a complex type definition:

- Section 34.5.1, “Occurrence Constraints on the All Element”
- Section 34.5.2, “Occurrence Constraints on the Choice Element”

- Section 34.5.3, “Occurrence Constraints on Elements”
- Section 34.5.4, “Occurrence Constraints on Sequences”

34.5.1. Occurrence Constraints on the All Element

XML Schema

Complex types defined with the **all** element do not allow for multiple occurrences of the structure defined by the **all** element. You can, however, make the structure defined by the **all** element optional by setting its **minOccurs** attribute to **0**.

Mapping to Java

Setting the **all** element's **minOccurs** attribute to **0** has no effect on the generated Java class.

34.5.2. Occurrence Constraints on the Choice Element

Overview

By default, the results of a **choice** element can only appear once in an instance of a complex type. You can change the number of times the element chosen to represent the structure defined by a **choice** element is allowed to appear using its **minOccurs** attribute and its **mxOccurs** attribute. Using these attributes you can specify that the choice type can occur zero to an unlimited number of times in an instance of a complex type. The element chosen for the choice type does not need to be the same for each occurrence of the type.

Using in XML Schema

The **minOccurs** attribute specifies the minimum number of times the choice type must appear. Its value can be any positive integer. Setting the **minOccurs** attribute to **0** specifies that the choice type does not need to appear inside an instance of the complex type.

The **maxOccurs** attribute specifies the maximum number of times the choice type can appear. Its value can be any non-zero, positive integer or **unbounded**. Setting the **maxOccurs** attribute to **unbounded** specifies that the choice type can appear an infinite number of times.

Example 34.18, “Choice Occurrence Constraints” shows the definition of a choice type, ClubEvent, with choice occurrence constraints. The choice type overall can be repeated 0 to unbounded times.

Example 34.18. Choice Occurrence Constraints

```
<complexType name="ClubEvent">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="MemberName" type="xsd:string"/>
    <element name="GuestName" type="xsd:string"/>
  </choice>
</complexType>
```

Mapping to Java

Unlike single instance choice structures, XML Schema choice structures that can occur multiple times are mapped to a Java class with a single member variable. This single member variable is a **List<T>** object that holds all of the data for the multiple occurrences of the sequence. For example, if the sequence defined in Example 34.18, "Choice Occurrence Constraints" occurred two times, then the list would have two items.

The name of the Java class' member variable is derived by concatenating the names of the member elements. The element names are separated by **Or** and the first letter of the variable name is converted to lower case. For example, the member variable generated from Example 34.18, "Choice Occurrence Constraints" would be named **memberNameOrGuestName**.

The type of object stored in the list depends on the relationship between the types of the member elements. For example:

- If the member elements are of the same type the generated list will contain **JAXBElement<T>** objects. The base type of the **JAXBElement<T>** objects is determined by the normal mapping of the member elements' type.
- If the member elements are of different types and their Java representations implement a common interface, the list will contains objects of the common interface.
- If the member elements are of different types and their Java representations extend a common base class, the list will contains objects of the common base class.
- If none of the other conditions are met, the list will contain **Object** objects.

The generated Java class will only have a getter method for the member variable. The getter method returns a reference to the live list. Any modifications made to the returned list will effect the actual object.

The Java class is decorated with the **@XmlType** annotation. The annotation's name property is set to the value of the **name** attribute from the parent element of the XML Schema definition. The annotation's propOrder property contains the single member variable representing the elements in the sequence.

The member variable representing the elements in the choice structure are decorated with the **@XmlElements** annotation. The **@XmlElements** annotation contains a comma separated list of **@XmlElement** annotations. The list has one **@XmlElement** annotation for each member element defined in the XML Schema definition of the type. The **@XmlElement** annotations in the list have their name property set to the value of the XML Schema **element** element's **name** attribute and their type property set to the Java class resulting from the mapping of the XML Schema **element** element's type.

Example 34.19, "Java Representation of Choice Structure with an Occurrence Constraint" shows the Java mapping for the XML Schema choice structure defined in Example 34.18, "Choice Occurrence Constraints".

Example 34.19. Java Representation of Choice Structure with an Occurrence Constraint

```
@XmlType(name = "ClubEvent", propOrder = {
    "memberNameOrGuestName"
})
public class ClubEvent {

    @XmlElementRefs({
        @XmlElementRef(name = "GuestName", type = JAXBElement.class),
        @XmlElementRef(name = "MemberName", type = JAXBElement.class)
    })
```

```
    protected List<JAXBElement<String>> memberNameOrGuestName;

    public List<JAXBElement<String>> getMemberNameOrGuestName() {
        if (memberNameOrGuestName == null) {
            memberNameOrGuestName = new ArrayList<JAXBElement<String>>
();
        }
        return this.memberNameOrGuestName;
    }

}
```

minOccurs set to 0

If only the **`minOccurs`** element is specified and its value is **`0`**, the code generators generate the Java class as if the **`minOccurs`** attribute were not set.

34.5.3. Occurrence Constraints on Elements

Overview

You can specify how many times a specific element in a complex type appears using the **`element`** element's **`minOccurs`** attribute and **`maxOccurs`** attribute. The default value for both attributes is **`1`**.

minOccurs set to 0

When you set one of the complex type's member element's **`minOccurs`** attribute to **`0`**, the **`@XmlElement`** annotation decorating the corresponding Java member variable is changed. Instead of having its required property set to **`true`**, the **`@XmlElement`** annotation's required property is set to **`false`**.

minOccurs set to a value greater than 1

In XML Schema you can specify that an element must occur more than once in an instance of the type by setting the **`element`** element's **`minOccurs`** attribute to a value greater than one. However, the generated Java class will not support the XML Schema constraint. Apache CXF generates the supporting Java member variable as if the **`minOccurs`** attribute were not set.

Elements with maxOccurs set

When you want a member element to appear multiple times in an instance of a complex type, you set the element's **`maxOccurs`** attribute to a value greater than 1. You can set the **`maxOccurs`** attribute's value to **unbounded** to specify that the member element can appear an unlimited number of times.

The code generators map a member element with the **`maxOccurs`** attribute set to a value greater than 1 to a Java member variable that is a **`List<T>`** object. The base class of the list is determined by mapping the element's type to Java. For XML Schema primitive types, the wrapper classes are used as described in the section called “Wrapper classes”. For example, if the member element is of type xsd:int the generated member variable is a **`List<Integer>`** object.

34.5.4. Occurrence Constraints on Sequences

Overview

By default, the contents of a **sequence** element can only appear once in an instance of a complex type. You can change the number of times the sequence of elements defined by a **sequence** element is allowed to appear using its **minOccurs** attribute and its **maxOccurs** attribute. Using these attributes you can specify that the sequence type can occur zero to an unlimited number of times in an instance of a complex type.

Using XML Schema

The **minOccurs** attribute specifies the minimum number of times the sequence must occur in an instance of the defined complex type. Its value can be any positive integer. Setting the **minOccurs** attribute to **0** specifies that the sequence does not need to appear inside an instance of the complex type.

The **maxOccurs** attribute specifies the upper limit for how many times the sequence can occur in an instance of the defined complex type. Its value can be any non-zero, positive integer or **unbounded**. Setting the **maxOccurs** attribute to **unbounded** specifies that the sequence can appear an infinite number of times.

Example 34.20, “Sequence with Occurrence Constraints” shows the definition of a sequence type, CultureInfo, with sequence occurrence constraints. The sequence can be repeated 0 to 2 times.

Example 34.20. Sequence with Occurrence Constraints

```
<complexType name="CultureInfo">
  <sequence minOccurs="0" maxOccurs="2">
    <element name="Name" type="string"/>
    <element name="Lcid" type="int"/>
  </sequence>
</complexType>
```

Mapping to Java

Unlike single instance sequences, XML Schema sequences that can occur multiple times are mapped to a Java class with a single member variable. This single member variable is a **List<T>** object that holds all of the data for the multiple occurrences of the sequence. For example, if the sequence defined in Example 34.20, “Sequence with Occurrence Constraints” occurred two times, then the list would have four items.

The name of the Java class' member variable is derived by concatenating the names of the member elements. The element names are separated by **And** and the first letter of the variable name is converted to lower case. For example, the member variable generated from Example 34.20, “Sequence with Occurrence Constraints” is named **nameAndLcid**.

The type of object stored in the list depends on the relationship between the types of the member elements. For example:

- If the member elements are of the same type the generated list will contain **JAXBElement<T>** objects. The base type of the **JAXBElement<T>** objects is determined by the normal mapping of the member elements' type.

- If the member elements are of different types and their Java representations implement a common interface, the list will contains objects of the common interface.
- If the member elements are of different types and their Java representations extend a common base class, the list will contain objects of the common base class.
- If none of the other conditions are met, the list will contain **`Object`** objects.

The generated Java class only has a getter method for the member variable. The getter method returns a reference to the live list. Any modifications made to the returned list effects the actual object.

The Java class is decorated with the **`@XmlType`** annotation. The annotation's name property is set to the value of the **`name`** attribute from the parent element of the XML Schema definition. The annotation's propOrder property contains the single member variable representing the elements in the sequence.

The member variable representing the elements in the sequence are decorated with the **`@XmlElements`** annotation. The **`@XmlElements`** annotation contains a comma separated list of **`@XmlElement`** annotations. The list has one **`@XmlElement`** annotation for each member element defined in the XML Schema definition of the type. The **`@XmlElement`** annotations in the list have their name property set to the value of the XML Schema **`element`** element's **`name`** attribute and their type property set to the Java class resulting from the mapping of the XML Schema **`element`** element's type.

Example 34.21, "Java Representation of Sequence with an Occurrence Constraint" shows the Java mapping for the XML Schema sequence defined in Example 34.20, "Sequence with Occurrence Constraints".

Example 34.21. Java Representation of Sequence with an Occurrence Constraint

```
@XmlType(name = "CultureInfo", propOrder = {
    "nameAndLcid"
})
public class CultureInfo {

    @XmlElements({
        @XmlElement(name = "Name", type = String.class),
        @XmlElement(name = "Lcid", type = Integer.class)
    })
    protected List<Serializable> nameAndLcid;

    public List<Serializable> getNameAndLcid() {
        if (nameAndLcid == null) {
            nameAndLcid = new ArrayList<Serializable>();
        }
        return this.nameAndLcid;
    }

}
```

minOccurs set to 0

If only the **`minOccurs`** element is specified and its value is **`0`**, the code generators generate the Java class as if the **`minOccurs`** attribute is not set.

34.6. USING MODEL GROUPS

Overview

XML Schema model groups are convenient shortcuts that allows you to reference a group of elements from a user-defined complex type.For example, you can define a group of elements that are common to several types in your application and then reference the group repeatedly. Model groups are defined using the **group** element, and are similar to complex type definitions. The mapping of model groups to Java is also similar to the mapping for complex types.

Defining a model group in XML Schema

You define a model group in XML Schema using the **group** element with the **name** attribute. The value of the **name** attribute is a string that is used to refer to the group throughout the schema. The **group** element, like the **complexType** element, can have the **sequence** element, the **all** element, or the **choice** element as its immediate child.

Inside the child element, you define the members of the group using **element** elements. For each member of the group, specify one **element** element. Group members can use any of the standard attributes for the **element** element including **minOccurs** and **maxOccurs**. So, if your group has three elements and one of them can occur up to three times, you define a group with three **element** elements, one of which uses maxOccurs="3". Example 34.22, “XML Schema Model Group” shows a model group with three elements.

Example 34.22. XML Schema Model Group

```
<group name="passenger">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="clubNum" type="xsd:long" />
    <element name="seatPref" type="xsd:string"
             maxOccurs="3" />
  </sequence>
</group>
```

Using a model group in a type definition

Once a model group has been defined, it can be used as part of a complex type definition. To use a model group in a complex type definition, use the **group** element with the **ref** attribute. The value of the **ref** attribute is the name given to the group when it was defined. For example, to use the group defined in Example 34.22, “XML Schema Model Group” you use <group ref="tns:passenger" /> as shown in Example 34.23, “Complex Type with a Model Group” .

Example 34.23. Complex Type with a Model Group

```
<complexType name="reservation">
  <sequence>
    <group ref="tns:passenger" />
    <element name="origin" type="xsd:string" />
    <element name="destination" type="xsd:string" />
```

```
    <element name="fltNum" type="xsd:long" />
  </sequence>
</complexType>
```

When a model group is used in a type definition, the group becomes a member of the type. So an instance of reservation has four member elements. The first element is the **passenger** element and it contains the member elements defined by the group shown in Example 34.22, "XML Schema Model Group". An example of an instance of reservation is shown in Example 34.24, "Instance of a Type with a Model Group".

Example 34.24. Instance of a Type with a Model Group

```
<reservation>
  <passenger>
    <name>A. Smart</name>
    <clubNum>99</clubNum>
    <seatPref>isle1</seatPref>
  </passenger>
  <origin>LAX</origin>
  <destination>FRA</destination>
  <fltNum>34567</fltNum>
</reservation>
```

Mapping to Java

By default, a model group is only mapped to Java artifacts when it is included in a complex type definition. When generating code for a complex type that includes a model group, Apache CXF simply includes the member variables for the model group into the Java class generated for the type. The member variables representing the model group are annotated based on the definitions of the model group.

Example 34.25, "Type with a Group" shows the Java class generated for the complex type defined in Example 34.23, "Complex Type with a Model Group" .

Example 34.25. Type with a Group

```
@XmlType(name = "reservation", propOrder = {
    "name",
    "clubNum",
    "seatPref",
    "origin",
    "destination",
    "fltNum"
})
public class Reservation {

    @XmlElement(required = true)
    protected String name;
    protected long clubNum;
    @XmlElement(required = true)
    protected List<String> seatPref;
    @XmlElement(required = true)
```

```
    protected String origin;
    @XmlElement(required = true)
    protected String destination;
    protected long fltNum;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public long getClubNum() {
        return clubNum;
    }

    public void setClubNum(long value) {
        this.clubNum = value;
    }

    public List<String> getSeatPref() {
        if (seatPref == null) {
            seatPref = new ArrayList<String>();
        }
        return this.seatPref;
    }

    public String getOrigin() {
        return origin;
    }

    public void setOrigin(String value) {
        this.origin = value;
    }

    public String getDestination() {
        return destination;
    }

    public void setDestination(String value) {
        this.destination = value;
    }

    public long getFltNum() {
        return fltNum;
    }

    public void setFltNum(long value) {
        this.fltNum = value;
    }
```

Multiple occurrences

You can specify that the model group appears more than once by setting the **group** element's **maxOccurs** attribute to a value greater than one. To allow for multiple occurrences of the model group Apache CXF maps the model group to a **List<T>** object. The **List<T>** object is generated following the rules for the group's first child:

- If the group is defined using a **sequence** element see Section 34.5.4, “Occurrence Constraints on Sequences”.
- If the group is defined using a **choice** element see Section 34.5.2, “Occurrence Constraints on the Choice Element”.

CHAPTER 35. USING WILD CARD TYPES

Abstract

There are instances when a schema author wants to defer binding elements or attributes to a defined type. For these cases, XML Schema provides three mechanisms for specifying wild card place holders. These are all mapped to Java in ways that preserve their XML Schema functionality.

35.1. USING ANY ELEMENTS

Overview

The XML Schema **any** element is used to create a wild card place holder in complex type definitions. When an XML element is instantiated for an XML Schema **any** element, it can be any valid XML element. The **any** element does not place any restrictions on either the content or the name of the instantiated XML element.

For example, given the complex type defined in Example 35.1, "XML Schema Type Defined with an Any Element" you can instantiate either of the XML elements shown in Example 35.2, "XML Document with an Any Element".

Example 35.1. XML Schema Type Defined with an Any Element

```
<element name="FlyBoy">
  <complexType>
    <sequence>
      <any />
      <element name="rank" type="xsd:int" />
    </sequence>
  </complexType>
</element>
```

Example 35.2. XML Document with an Any Element

```
<FlyBoy>
  <learJet>CL-215</learJet>
  <rank>2</rank>
</element>
<FlyBoy>
  <viper>Mark II</viper>
  <rank>1</rank>
</element>
```

XML Schema **any** elements are mapped to either a Java `Object` object or a Java `org.w3c.dom.Element` object.

Specifying in XML Schema

The **any** element can be used when defining sequence complex types and choice complex types. In most cases, the **any** element is an empty element. It can, however, take an **annotation** element as a child.

Table 35.1, "Attributes of the XML Schema Any Element" describes the **any** element's attributes.

Table 35.1. Attributes of the XML Schema Any Element

Attribute	Description
namespace	Specifies the namespace of the elements that can be used to instantiate the element in an XML document. The valid values are: **##any** Specifies that elements from any namespace can be used. This is the default. **##other** Specifies that elements from any namespace *other than the parent element's namespace*can be used. **##local** Specifies elements without a namespace must be used. **##targetNamespace** Specifies that elements from the parent element's namespace must be used. **A space delimited list of URIs,##local, and ##targetNamespace** Specifies that elements from any of the listed namespaces can be used.
maxOccurs	Specifies the maximum number of times an instance of the element can appear in the parent element. The default value is **1**. To specify that an instance of the element can appear an unlimited number of times, you can set the attribute's value to **unbounded**.
minOccurs	Specifies the minimum number of times an instance of the element can appear in the parent element. The default value is **1**.

Attribute	Description
processContents	Specifies how the element used to instantiate the any element should be validated. Valid values are: **strict** Specifies that the element must be validated against the proper schema. This is the default value. **lax** Specifies that the element should be validated against the proper schema. If it cannot be validated, no errors are thrown. **skip** Specifies that the element should not be validated.

Example 35.3, “Complex Type Defined with an Any Element” shows a complex type defined with an **any** element

Example 35.3. Complex Type Defined with an Any Element

```
<complexType name="surprisePackage">
  <sequence>
    <any processContents="lax" />
    <element name="to" type="xsd:string" />
    <element name="from" type="xsd:string" />
  </sequence>
</complexType>
```

Mapping to Java

XML Schema **any** elements result in the creation of a Java property named **any**. The property has associated getter and setter methods. The type of the resulting property depends on the value of the element's **processContents** attribute. If the **any** element's **processContents** attribute is set to **skip**, the element is mapped to a **org.w3c.dom.Element** object. For all other values of the **processContents** attribute an **any** element is mapped to a Java **Object** object.

The generated property is decorated with the **@XmlAnyElement** annotation. This annotation has an optional lax property that instructs the runtime what to do when marshaling the data. Its default value is **false** which instructs the runtime to automatically marshal the data into a **org.w3c.dom.Element** object. Setting lax to **true** instructs the runtime to attempt to marshal the data into JAXB types. When the **any** element's **processContents** attribute is set to **skip**, the lax property is set to its default value. For all other values of the **processContents** attribute, lax is set to **true**.

Example 35.4, “Java Class with an Any Element” shows how the complex type defined in Example 35.3, “Complex Type Defined with an Any Element” is mapped to a Java class.

Example 35.4. Java Class with an Any Element

```
public class SurprisePackage {

    @XmlAnyElement(lax = true)
    protected Object any;
    @XmlElement(required = true)
    protected String to;
    @XmlElement(required = true)
    protected String from;

    public Object getAny() {
        return any;
    }

    public void setAny(Object value) {
        this.any = value;
    }

    public String getTo() {
        return to;
    }

    public void setTo(String value) {
        this.to = value;
    }

    public String getFrom() {
        return from;
    }

    public void setFrom(String value) {
        this.from = value;
    }

}
```

Marshalling

If the Java property for an **any** element has its lax set to **false**, or the property is not specified, the runtime makes no attempt to parse the XML data into JAXB objects. The data is always stored in a DOM **Element** object.

If the Java property for an **any** element has its lax set to **true**, the runtime attempts to marshal the XML data into the appropriate JAXB objects. The runtime attempts to identify the proper JAXB classes using the following procedure:

1. It checks the element tag of the XML element against the list of elements known to the runtime. If it finds a match, the runtime marshals the XML data into the proper JAXB class for the element.

2. It checks the XML element's **xsi:type** attribute. If it finds a match, the runtime marshals the XML element into the proper JAXB class for that type.

3. If it cannot find a match it marshals the XML data into a DOM **Element** object.

Usually an application's runtime knows about all of the types generated from the schema's included in its contract. This includes the types defined in the contract's **wsdl:types** element, any data types added to the contract through inclusion, and any types added to the contract through importing other schemas. You can also make the runtime aware of additional types using the **@XmlSeeAlso** annotation which is described in Section 31.4, “Adding Classes to the Runtime Marshaller” .

Unmarshalling

If the Java property for an **any** element has its lax set to **false**, or the property is not specified, the runtime will only accept DOM **Element** objects. Attempting to use any other type of object will result in a marshalling error.

If the Java property for an **any** element has its lax set to **true**, the runtime uses its internal map between Java data types and the XML Schema constructs they represent to determine the XML structure to write to the wire. If the runtime knows the class and can map it to an XML Schema construct, it writes out the data and inserts an **xsi:type** attribute to identify the type of data the element contains.

If the runtime cannot map the Java object to a known XML Schema construct, it will throw a marshaling exception. You can add types to the runtime's map using the **@XmlSeeAlso** annotation which is described in Section 31.4, “Adding Classes to the Runtime Marshaller” .

35.2. USING THE XML SCHEMA ANYTYPE TYPE

Overview

The XML Schema type xsd:anyType is the root type for all XML Schema types. All of the primitives are derivatives of this type, as are all user defined complex types. As a result, elements defined as being of xsd:anyType can contain data in the form of any of the XML Schema primitives as well as any complex type defined in a schema document.

In Java the closest matching type is the **Object** class. It is the class from which all other Java classes are sub-typed.

Using in XML Schema

You use the xsd:anyType type as you would any other XML Schema complex type. It can be used as the value of an **element** element's **type** element. It can also be used as the base type from which other types are defined.

Example 35.5, “Complex Type with a Wild Card Element” shows an example of a complex type that contains an element of type xsd:anyType.

Example 35.5. Complex Type with a Wild Card Element

```
<complexType name="wildStar">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="ship" type="xsd:anyType" />
  </sequence>
</complexType>
```

Mapping to Java

Elements that are of type xsd:anyType are mapped to **`Object`** objects. Example 35.6, "Java Representation of a Wild Card Element" shows the mapping of Example 35.5, "Complex Type with a Wild Card Element" to a Java class.

Example 35.6. Java Representation of a Wild Card Element

```
public class WildStar {

    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected Object ship;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public Object getShip() {
        return ship;
    }

    public void setShip(Object value) {
        this.ship = value;
    }
}
```

This mapping allows you to place any data into the property representing the wild card element. The Apache CXF runtime handles the marshaling and unmarshaling of the data into usable Java representation.

Marshalling

When Apache CXF marshals XML data into Java types, it attempts to marshal anyType elements into known JAXB objects. To determine if it is possible to marshal an anyType element into a JAXB generated object, the runtime inspects the element's **`xsi:type`** attribute to determine the actual type used to construct the data in the element. If the **`xsi:type`** attribute is not present, the runtime attempts to identify the element's actual data type by introspection. If the element's actual data type is determined to be one of the types known by the application's JAXB context, the element is marshaled into a JAXB object of the proper type.

If the runtime cannot determine the actual data type of the element, or the actual data type of the element is not a known type, the runtime marshals the content into a **`org.w3c.dom.Element`** object. You will then need to work with the element's content using the DOM APIs.

An application's runtime usually knows about all of the types generated from the schema's included in its contract. This includes the types defined in the contract's **`wsdl:types`** element, any data types

added to the contract through inclusion, and any types added to the contract through importing other schema documents. You can also make the runtime aware of additional types using the **@XmlSeeAlso** annotation which is described in Section 31.4, “Adding Classes to the Runtime Marshaller” .

Unmarshalling

When Apache CXF unmarshals Java types into XML data, it uses an internal map between Java data types and the XML Schema constructs they represent to determine the XML structure to write to the wire. If the runtime knows the class and can map the class to an XML Schema construct, it writes out the data and inserts an **xsi:type** attribute to identify the type of data the element contains. If the data is stored in a **org.w3c.dom.Element** object, the runtime writes the XML structure represented by the object but it does not include an **xsi:type** attribute.

If the runtime cannot map the Java object to a known XML Schema construct, it throws a marshaling exception. You can add types to the runtime's map using the **@XmlSeeAlso** annotation which is described in Section 31.4, “Adding Classes to the Runtime Marshaller” .

35.3. USING UNBOUND ATTRIBUTES

Overview

XML Schema has a mechanism that allows you to leave a place holder for an arbitrary attribute in a complex type definition. Using this mechanism, you can define a complex type that can have any attribute. For example, you can create a type that defines the elements <robot name="epsilon" />, <robot age="10000" />, or <robot type="weevil" /> without specifying the three attributes. This can be particularly useful when flexibility in your data is required.

Defining in XML Schema

Undeclared attributes are defined in XML Schema using the **anyAttribute** element. It can be used wherever an attribute element can be used. The **anyAttribute** element has no attributes, as shown in Example 35.7, “Complex Type with an Undeclared Attribute” .

Example 35.7. Complex Type with an Undeclared Attribute

```
<complexType name="arbitter">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="rate" type="xsd:float" />
  </sequence>
  <anyAttribute />
</complexType>
```

The defined type, arbitter, has two elements and can have one attribute of any type. The elements three elements shown in Example 35.8, “Examples of Elements Defined with a Wild Card Attribute” can all be generated from the complex type arbitter.

Example 35.8. Examples of Elements Defined with a Wild Card Attribute

```
<officer rank="12"><name>...</name><rate>...</rate></officer>
<lawyer type="divorce"><name>...</name><rate>...</rate></lawyer>
<judge><name>...</name><rate>...</rate></judge>
```

Mapping to Java

When a complex type containing an **anyAttribute** element is mapped to Java, the code generator adds a member called **otherAttributes** to the generated class. **otherAttributes** is of type **java.util.Map<QName, String>** and it has a getter method that returns a live instance of the map. Because the map returned from the getter is live, any modifications to the map are automatically applied. Example 35.9, "Class for a Complex Type with an Undeclared Attribute" shows the class generated for the complex type defined in Example 35.7, "Complex Type with an Undeclared Attribute".

Example 35.9. Class for a Complex Type with an Undeclared Attribute

```
public class Arbitter {

    @XmlElement(required = true)
    protected String name;
    protected float rate;

    @XmlAnyAttribute
    private Map<QName, String> otherAttributes = new HashMap<QName,
String>();

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public float getRate() {
        return rate;
    }

    public void setRate(float value) {
        this.rate = value;
    }

    public Map<QName, String> getOtherAttributes() {
        return otherAttributes;
    }

}
```

Working with undeclared attributes

The **otherAttributes** member of the generated class expects to be populated with a **Map** object. The map is keyed using **QNames**. Once you get the map , you can access any attributes set on the object and set new attributes on the object.

Example 35.10, "Working with Undeclared Attributes" shows sample code for working with undeclared

attributes.

Example 35.10. Working with Undeclared Attributes

```
Arbitter judge = new Arbitter();
1 Map<QName, String> otherAtts = judge.getOtherAttributes();

2 QName at1 = new QName("test.apache.org", "house");
  QName at2 = new QName("test.apache.org", "veteran");

3 otherAtts.put(at1, "Cape");
  otherAtts.put(at2, "false");

4 String vetStatus = otherAtts.get(at2);
```

The code in Example 35.10, "Working with Undeclared Attributes" does the following:

1. Gets the map containing the undeclared attributes.
2. Creates QNames to work with the attributes.
3. Sets the values for the attributes into the map.
2. Retrieves the value for one of the attributes.

CHAPTER 36. ELEMENT SUBSTITUTION

Abstract

XML Schema substitution groups allow you to define a group of elements that can replace a top level, or head, element. This is useful in cases where you have multiple elements that share a common base type or with elements that need to be interchangeable.

36.1. SUBSTITUTION GROUPS IN XML SCHEMA

Overview

A substitution group is a feature of XML schema that allows you to specify elements that can replace another element in documents generated from that schema. The replaceable element is called the head element and must be defined in the schema's global scope. The elements of the substitution group must be of the same type as the head element or a type that is derived from the head element's type.

In essence, a substitution group allows you to build a collection of elements that can be specified using a generic element. For example, if you are building an ordering system for a company that sells three types of widgets you might define a generic widget element that contains a set of common data for all three widget types. Then you can define a substitution group that contains a more specific set of data for each type of widget. In your contract you can then specify the generic widget element as a message part instead of defining a specific ordering operation for each type of widget. When the actual message is built, the message can contain any of the elements of the substitution group.

Syntax

Substitution groups are defined using the **substitutionGroup** attribute of the XML Schema **element** element. The value of the **substitutionGroup** attribute is the name of the element that the element being defined replaces. For example, if your head element is **widget**, adding the attribute substitutionGroup="widget" to an element named **woodWidget** specifies that anywhere a **widget** element is used, you can substitute a **woodWidget** element. This is shown in Example 36.1, "Using a Substitution Group".

Example 36.1. Using a Substitution Group

```
<element name="widget" type="xsd:string" />
<element name="woodWidget" type="xsd:string"
         substitutionGroup="widget" />
```

Type restrictions

The elements of a substitution group must be of the same type as the head element or of a type derived from the head element's type. For example, if the head element is of type xsd:int all members of the substitution group must be of type xsd:int or of a type derived from xsd:int. You can also define a substitution group similar to the one shown in Example 36.2, "Substitution Group with Complex Types" where the elements of the substitution group are of types derived from the head element's type.

Example 36.2. Substitution Group with Complex Types

```
<complexType name="widgetType">
  <sequence>
    <element name="shape" type="xsd:string" />
    <element name="color" type="xsd:string" />
  </sequence>
</complexType>
<complexType name="woodWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="woodType" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="plasticWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="moldProcess" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="widget" type="widgetType" />
<element name="woodWidget" type="woodWidgetType"
         substitutionGroup="widget" />
<element name="plasticWidget" type="plasticWidgetType"
         substitutionGroup="widget" />
<complexType name="partType">
  <sequence>
    <element ref="widget" />
  </sequence>
</complexType>
<element name="part" type="partType" />
```

The head element of the substitution group, **widget**, is defined as being of type widgetType. Each element of the substitution group extends widgetType to include data that is specific to ordering that type of widget.

Based on the schema in Example 36.2, “Substitution Group with Complex Types” , the **part** elements in Example 36.3, “XML Document using a Substitution Group” are valid.

Example 36.3. XML Document using a Substitution Group

```
<part>
  <widget>
    <shape>round</shape>
    <color>blue</color>
  </widget>
</part>
<part>
  <plasticWidget>
    <shape>round</shape>
```

```
    <color>blue</color>
    <moldProcess>sandCast</moldProcess>
  </plasticWidget>
</part>
<part>
  <woodWidget>
    <shape>round</shape>
    <color>blue</color>
    <woodType>elm</woodType>
  </woodWidget>
</part>
```

Abstract head elements

You can define an abstract head element that can never appear in a document produced using your schema. Abstract head elements are similar to abstract classes in Java because they are used as the basis for defining more specific implementations of a generic class. Abstract heads also prevent the use of the generic element in the final product.

You declare an abstract head element by setting the **`abstract`** attribute of an **`element`** element to **`true`**, as shown in Example 36.4, "Abstract Head Definition". Using this schema, a valid **`review`** element can contain either a **`positiveComment`** element or a **`negativeComment`** element, but cannot contain a **`comment`** element.

Example 36.4. Abstract Head Definition

```
<element name="comment" type="xsd:string" abstract="true" />
<element name="positiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="negtiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="review">
  <complexContent>
    <all>
      <element name="custName" type="xsd:string" />
      <element name="impression" ref="comment" />
    </all>
  </complexContent>
</element>
```

36.2. SUBSTITUTION GROUPS IN JAVA

Overview

Apache CXF, as specified in the JAXB specification, supports substitution groups using Java's native class hierarchy in combination with the ability of the **`JAXBElement`** class' support for wildcard definitions. Because the members of a substitution group must all share a common base type, the classes generated to support the elements' types also share a common base type. In addition, Apache CXF maps instances of the head element to **`JAXBElement<? extends T>`** properties.

Generated object factory methods

The object factory generated to support a package containing a substitution group has methods for each of the elements in the substitution group. For each of the members of the substitution group, except for the head element, the `@XmlElementDecl` annotation decorating the object factory method includes two additional properties, as described in Table 36.1, "Properties for Declaring a JAXB Element is a Member of a Substitution Group".

Table 36.1. Properties for Declaring a JAXB Element is a Member of a Substitution Group

Property	Description
substitutionHeadNamespace	Specifies the namespace where the head element is defined.
substitutionHeadName	Specifies the value of the head element's `name` attribute.

The object factory method for the head element of the substitution group's `@XmlElementDecl` contains only the default namespace property and the default name property.

In addition to the element instantiation methods, the object factory contains a method for instantiating an object representing the head element. If the members of the substitution group are all of complex types, the object factory also contains methods for instantiating instances of each complex type used.

Example 36.5, "Object Factory Method for a Substitution Group" shows the object factory method for the substitution group defined in Example 36.2, "Substitution Group with Complex Types" .

Example 36.5. Object Factory Method for a Substitution Group

```
public class ObjectFactory {

    private final static QName _Widget_QNAME = new QName(...);
    private final static QName _PlasticWidget_QNAME = new QName(...);
    private final static QName _WoodWidget_QNAME = new QName(...);

    public ObjectFactory() {
    }

    public WidgetType createWidgetType() {
        return new WidgetType();
    }

    public PlasticWidgetType createPlasticWidgetType() {
        return new PlasticWidgetType();
    }

    public WoodWidgetType createWoodWidgetType() {
        return new WoodWidgetType();
    }

    @XmlElementDecl(namespace="...", name = "widget")
    public JAXBElement<WidgetType> createWidget(WidgetType value) {
        return new JAXBElement<WidgetType>(_Widget_QNAME,
```

```
WidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "plasticWidget",
substitutionHeadNamespace = "...", substitutionHeadName = "widget")
    public JAXBElement<PlasticWidgetType>
createPlasticWidget(PlasticWidgetType value) {
        return new JAXBElement<PlasticWidgetType>(_PlasticWidget_QNAME,
PlasticWidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "woodWidget",
substitutionHeadNamespace = "...", substitutionHeadName = "widget")
    public JAXBElement<WoodWidgetType> createWoodWidget(WoodWidgetType
value) {
        return new JAXBElement<WoodWidgetType>(_WoodWidget_QNAME,
WoodWidgetType.class, null, value);
    }

}
```

Substitution groups in interfaces

If the head element of a substitution group is used as a message part in one of an operation's messages, the resulting method parameter will be an object of the class generated to support that element. It will not necessarily be an instance of the **`JAXBElement<? extends T>`** class. The runtime relies on Java's native type hierarchy to support the type substitution, and Java will catch any attempts to use unsupported types.

To ensure that the runtime knows all of the classes needed to support the element substitution, the SEI is decorated with the **`@XmlSeeAlso`** annotation. This annotation specifies a list of classes required by the runtime for marshalling. Fore more information on using the **`@XmlSeeAlso`** annotation see Section 31.4, “Adding Classes to the Runtime Marshaller” .

Example 36.7, “Generated Interface Using a Substitution Group” shows the SEI generated for the interface shown in Example 36.6, “WSDL Interface Using a Substitution Group” . The interface uses the substitution group defined in Example 36.2, “Substitution Group with Complex Types” .

Example 36.6. WSDL Interface Using a Substitution Group

```
<message name="widgetMessage">
    <part name="widgetPart" element="xsd1:widget" />
  </message>
  <message name="numWidgets">
    <part name="numInventory" type="xsd:int" />
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int" />
  </message>
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order" />
      <output message="tns:widgetOrderBill" name="bill" />
      <fault message="tns:badSize" name="sizeFault" />
```

```
    </operation>
    <operation name="checkWidgets">
      <input message="tns:widgetMessage" name="request" />
      <output message="tns:numWidgets" name="response" />
    </operation>
  </portType>
```

Example 36.7. Generated Interface Using a Substitution Group

```
@WebService(targetNamespace = "...", name = "orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName =
"numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget",
targetNamespace = "...")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );
}
```

TIP

The SEI shown in Example 36.7, “Generated Interface Using a Substitution Group” lists the object factory in the **@XmlSeeAlso** annotation. Listing the object factory for a namespace provides access to all of the generated classes for that namespace.

Substitution groups in complex types

When the head element of a substitution group is used as an element in a complex type, the code generator maps the element to a **JAXBElement<? extends T>** property. It does not map it to a property containing an instance of the generated class generated to support the substitution group.

For example, the complex type defined in Example 36.8, “Complex Type Using a Substitution Group” results in the Java class shown in Example 36.9, “Java Class for a Complex Type Using a Substitution Group”. The complex type uses the substitution group defined in Example 36.2, “Substitution Group with Complex Types”.

Example 36.8. Complex Type Using a Substitution Group

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element ref="xsd1:widget"/>
  </sequence>
</complexType>
```

Example 36.9. Java Class for a Complex Type Using a Substitution Group

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "widgetOrderInfo", propOrder = {"amount","widget",})
public class WidgetOrderInfo {

    protected int amount;
    @XmlElementRef(name = "widget", namespace = "...", type =
JAXBElement.class)
    protected JAXBElement<? extends WidgetType> widget;
    public int getAmount() {
        return amount;
    }

    public void setAmount(int value) {
        this.amount = value;
    }

    public JAXBElement<? extends WidgetType> getWidget() {
        return widget;
    }

    public void setWidget(JAXBElement<? extends WidgetType> value) {
        this.widget = ((JAXBElement<? extends WidgetType> ) value);
    }

}
```

Setting a substitution group property

How you work with a substitution group depends on whether the code generator mapped the group to a straight Java class or to a **JAXBElement<? extends T>** class. When the element is simply mapped to an object of the generated value class, you work with the object the same way you work with other Java objects that are part of a type hierarchy. You can substitute any of the subclasses for the parent class. You can inspect the object to determine its exact class, and cast it appropriately.

TIP

The JAXB specification recommends that you use the object factory methods for instantiating objects of the generated classes.

When the code generators create a **JAXBElement<? extends T>** object to hold instances of a substitution group, you must wrap the element's value in a **JAXBElement<? extends T>** object. The best method to do this is to use the element creation methods provided by the object factory. They provide an easy means for creating an element based on its value.

Example 36.10, “Setting a Member of a Substitution Group” shows code for setting an instance of a substitution group.

Example 36.10. Setting a Member of a Substitution Group

```
1 ObjectFactory of = new ObjectFactory();
```

```
(2) PlasticWidgetType pWidget = of.createPlasticWidgetType();
    pWidget.setShape = "round';
pWidget.setColor = "green";
pWidget.setMoldProcess = "injection";

JAXBElement<PlasticWidgetType> widget = of.createPlasticWidget(pWidget);
(3)
(4) WidgetOrderInfo order = of.createWidgetOrderInfo();
(5) order.setWidget(widget);
```

The code in Example 36.10, "Setting a Member of a Substitution Group" does the following:

1. Instantiates an object factory.
2. Instantiates a **`PlasticWidgetType`** object.
3. Instantiates a **`JAXBElement<PlasticWidgetType>`** object to hold a plastic widget element.
4. Instantiates a **`WidgetOrderInfo`** object.
5. Sets the **`WidgetOrderInfo`** object's widget to the **`JAXBElement`** object holding the plastic widget element.

Getting the value of a substitution group property

The object factory methods do not help when extracting the element's value from a **`JAXBElement<? extends T>`** object. You must to use the **`JAXBElement<? extends T>`** object's **`getValue()`** method. The following options determine the type of object returned by the **`getValue()`** method:

- Use the **`isInstance()`** method of all the possible classes to determine the class of the element's value object.
- Use the **`JAXBElement<? extends T>`** object's **`getName()`** method to determine the element's name.

  The **`getName()`** method returns a QName. Using the local name of the element, you can determine the proper class for the value object.
- Use the **`JAXBElement<? extends T>`** object's **`getDeclaredType()`** method to determine the class of the value object.

  The **`getDeclaredType()`** method returns the **`Class`** object of the element's value object.

WARNING

There is a possibility that the **`getDeclaredType()`** method will return the base class for the head element regardless of the actual class of the value object.

Example 36.11, "Getting the Value of a Member of the Substitution Group" shows code retrieving the value from a substitution group. To determine the proper class of the element's value object the example uses the element's **getName()** method.

Example 36.11. Getting the Value of a Member of the Substitution Group

```
String elementName = order.getWidget().getName().getLocalPart();
if (elementName.equals("woodWidget")
{
  WoodWidgetType widget=order.getWidget().getValue();
}
else if (elementName.equals("plasticWidget")
{
  PlasticWidgetType widget=order.getWidget().getValue();
}
else
{
  WidgetType widget=order.getWidget().getValue();
}
```

36.3. WIDGET VENDOR EXAMPLE

This section shows an example of substitution groups being used in Apache CXF to solve a real world application. A service and consumer are developed using the widget substitution group defined in Example 36.2, "Substitution Group with Complex Types" . The service offers two operations: **checkWidgets** and **placeWidgetOrder**. Example 36.12, "Widget Ordering Interface" shows the interface for the ordering service.

Example 36.12. Widget Ordering Interface

```
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation"
        type="xsd1:widgetOrderBillInfo"/>
</message>
<message name="widgetMessage">
  <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

Example 36.13, "Widget Ordering SEI" shows the generated Java SEI for the interface.

Example 36.13. Widget Ordering SEI

```
@WebService(targetNamespace = "http://widgetVendor.com/widgetOrderForm",
name = "orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName =
"numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget",
targetNamespace = "http://widgetVendor.com/types/widgetTypes")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "widgetOrderConformation", targetNamespace = "",
partName = "widgetOrderConformation")
    @WebMethod
    public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo
placeWidgetOrder(
        @WebParam(partName = "widgetOrderForm", name =
"widgetOrderForm", targetNamespace = "")
        com.widgetvendor.types.widgettypes.WidgetOrderInfo
widgetOrderForm
    ) throws BadSize;
}
```

NOTE

Because the example only demonstrates the use of substitution groups, some of the business logic is not shown.

36.3.1. The checkWidgets Operation

Overview

checkWidgets is a simple operation that has a parameter that is the head member of a substitution group. This operation demonstrates how to deal with individual parameters that are members of a substitution group. The consumer must ensure that the parameter is a valid member of the substitution group. The service must properly determine which member of the substitution group was sent in the request.

Consumer implementation

The generated method signature uses the Java class supporting the type of the substitution group's head element. Because the member elements of a substitution group are either of the same type as the

head element or of a type derived from the head element's type, the Java classes generated to support the members of the substitution group inherit from the Java class generated to support the head element. Java's type hierarchy natively supports using subclasses in place of the parent class.

Because of how Apache CXF generates the types for a substitution group and Java's type hierarchy, the client can invoke **checkWidgets()** without using any special code. When developing the logic to invoke **checkWidgets()** you can pass in an object of one of the classes generated to support the widget substitution group.

Example 36.14, "Consumer Invoking **checkWidgets()**" shows a consumer invoking **checkWidgets()**.

Example 36.14. Consumer Invoking checkWidgets()

```
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
  case '1':
  {
    WidgetType widget = new WidgetType();
    ...
    break;
  }
  case '2':
  {
    WoodWidgetType widget = new WoodWidgetType();
    ...
    break;
  }
  case '3':
  {
    PlasticWidgetType widget = new PlasticWidgetType();
    ...
    break;
  }
  default :
    System.out.println("Invaid Widget Selection!!");
}

proxy.checkWidgets(widgets);
```

Service implementation

The service's implementation of **checkWidgets()** gets a widget description as a **WidgetType** object, checks the inventory of widgets, and returns the number of widgets in stock. Because all of the classes used to implement the substitution group inherit from the same base class, you can implement **checkWidgets()** without using any JAXB specific APIs.

All of the classes generated to support the members of the substitution group for **`widget`** extend the **`WidgetType`** class. Because of this fact, you can use **`instanceof`** to determine what type of widget was passed in and simply cast the ***`widgetPart`*** object into the more restrictive type if appropriate. Once you have the proper type of object, you can check the inventory of the right kind of widget.

Example 36.15, "Service Implementation of **`checkWidgets()`**" shows a possible implementation.

Example 36.15. Service Implementation of `checkWidgets()`

```
public int checkWidgets(WidgetType widgetPart)
{
  if (widgetPart instanceof WidgetType)
  {
    return checkWidgetInventory(widgetType);
  }
  else if (widgetPart instanceof WoodWidgetType)
  {
    WoodWidgetType widget = (WoodWidgetType)widgetPart;
    return checkWoodWidgetInventory(widget);
  }
  else if (widgetPart instanceof PlasticWidgetType)
  {
    PlasticWidgetType widget = (PlasticWidgetType)widgetPart;
    return checkPlasticWidgetInventory(widget);
  }
}
```

36.3.2. The placeWidgetOrder Operation

Overview

`placeWidgetOrder` uses two complex types containing the substitution group. This operation demonstrates to use such a structure in a Java implementation. Both the consumer and the service must get and set members of a substitution group.

Consumer implementation

To invoke **`placeWidgetOrder()`** the consumer must construct a widget order containing one element of the widget substitution group. When adding the widget to the order, the consumer should use the object factory methods generated for each element of the substitution group. This ensures that the runtime and the service can correctly process the order. For example, if an order is being placed for a plastic widget, the **`ObjectFactory.createPlasticWidget()`** method is used to create the element before adding it to the order.

Example 36.16, "Setting a Substitution Group Member" shows consumer code for setting the widget property of the **`WidgetOrderInfo`** object.

Example 36.16. Setting a Substitution Group Member

```
ObjectFactory of = new ObjectFactory();

WidgetOrderInfo order = new of.createWidgetOrderInfo();
```

```
...
System.out.println();
System.out.println("What color widgets do you want to order?");
String color = reader.readLine();
System.out.println();
System.out.println("What shape widgets do you want to order?");
String shape = reader.readLine();
System.out.println();
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
  case '1':
  {
    WidgetType widget = of.createWidgetType();
    widget.setColor(color);
    widget.setShape(shape);
    JAXB<WidgetType> widgetElement = of.createWidget(widget);
    order.setWidget(widgetElement);
    break;
  }
  case '2':
  {
    WoodWidgetType woodWidget = of.createWoodWidgetType();
    woodWidget.setColor(color);
    woodWidget.setShape(shape);
    System.out.println();
    System.out.println("What type of wood are your widgets?");
    String wood = reader.readLine();
    woodWidget.setWoodType(wood);
    JAXB<WoodWidgetType> widgetElement =
of.createWoodWidget(woodWidget);
    order.setWoodWidget(widgetElement);
    break;
  }
  case '3':
  {
    PlasticWidgetType plasticWidget = of.createPlasticWidgetType();
    plasticWidget.setColor(color);
    plasticWidget.setShape(shape);
    System.out.println();
    System.out.println("What type of mold to use for your
                        widgets?");
    String mold = reader.readLine();
    plasticWidget.setMoldProcess(mold);
    JAXB<WidgetType> widgetElement =
of.createPlasticWidget(plasticWidget);
    order.setPlasticWidget(widgetElement);
    break;
  }
```

```
    default :
      System.out.println("Invaid Widget Selection!!");
      }
```

Service implementation

The **placeWidgetOrder()** method receives an order in the form of a **WidgetOrderInfo** object, processes the order, and returns a bill to the consumer in the form of a **WidgetOrderBillInfo** object. The orders can be for a plain widget, a plastic widget, or a wooden widget. The type of widget ordered is determined by what type of object is stored in ***widgetOrderForm*** object's widget property. The widget property is a substitution group and can contain a **widget** element, a **woodWidget** element, or a **plasticWidget** element.

The implementation must determine which of the possible elements is stored in the order. This can be accomplished using the **JAXBElement<? extends T>** object's **getName()** method to determine the element's QName. The QName can then be used to determine which element in the substitution group is in the order. Once the element included in the bill is known, you can extract its value into the proper type of object.

Example 36.17, "Implementation of **placeWidgetOrder()**" shows a possible implementation.

Example 36.17. Implementation of placeWidgetOrder()

```
public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo
placeWidgetOrder(WidgetOrderInfo widgetOrderForm)
{
1   ObjectFactory of = new ObjectFactory();

2   WidgetOrderBillInfo bill = new WidgetOrderBillInfo()

   // Copy the shipping address and the number of widgets
   // ordered from widgetOrderForm to bill
   ...

3   int numOrdered = widgetOrderForm.getAmount();

  String elementName =
4 widgetOrderForm.getWidget().getName().getLocalPart();
5    if (elementName.equals("woodWidget")
     {
6    WoodWidgetType widget=order.getWidget().getValue();
     buildWoodWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<WoodWidgetType> widgetElement =
7 of.createWoodWidget(widget);
8      bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.75;
9     bill.setAmountDue(amtDue);
    }
  else if (elementName.equals("plasticWidget")
  {
    PlasticWidgetType widget=order.getWidget().getValue();
```

```
    buildPlasticWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<PlasticWidgetType> widgetElement =
of.createPlasticWidget(widget);
    bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.90;
    bill.setAmountDue(amtDue);
  }
  else
  {
    WidgetType widget=order.getWidget().getValue();
    buildWidget(widget, numOrdered);

    // Add the widget info to bill
    JAXBElement<WidgetType> widgetElement = of.createWidget(widget);
    bill.setWidget(widgetElement);

    float amtDue = numOrdered * 0.30;
    bill.setAmountDue(amtDue);
  }

  return(bill);
}
```

The code in Example 36.17, "Implementation of **`placeWidgetOrder()`**" does the following:

1. Instantiates an object factory to create elements.
2. Instantiates a **`WidgetOrderBillInfo`** object to hold the bill.
3. Gets the number of widgets ordered.
4. Gets the local name of the element stored in the order.
5. Checks to see if the element is a **`woodWidget`** element.
6. Extracts the value of the element from the order to the proper type of object.
7. Creates a **`JAXBElement<T>`** object placed into the bill.
8. Sets the bill object's widget property.
9. Sets the bill object's amountDue property.

CHAPTER 37. CUSTOMIZING HOW TYPES ARE GENERATED

Abstract

The default JAXB mappings address most of the cases encountered when using XML Schema to define the objects for a Java application. For instances where the default mappings are insufficient, JAXB provides an extensive customization mechanism.

37.1. BASICS OF CUSTOMIZING TYPE MAPPINGS

Overview

The JAXB specification defines a number of XML elements that customize how Java types are mapped to XML Schema constructs. These elements can be specified in-line with XML Schema constructs. If you cannot, or do not want to, modify the XML Schema definitions, you can specify the customizations in external binding document.

Namespace

The elements used to customize the JAXB data bindings are defined in the namespace **`http://java.sun.com/xml/ns/jaxb`**. You must add a namespace declaration similar to the one shown in Example 37.1, "JAXB Customization Namespace". This is added to the root element of all XML documents defining JAXB customizations.

Example 37.1. JAXB Customization Namespace

```
xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
```

Version declaration

When using the JAXB customizations, you must indicate the JAXB version being used. This is done by adding a **`jaxb:version`** attribute to the root element of the external binding declaration. If you are using in-line customization, you must include the **`jaxb:version`** attribute in the **`schema`** element containing the customizations. The value of the attribute is always **`2.0`**.

Example 37.2, "Specifying the JAXB Customization Version" shows an example of the **`jaxb:version`** attribute used in a **`schema`** element.

Example 37.2. Specifying the JAXB Customization Version

```
< schema ...
        jaxb:version="2.0">
```

Using in-line customization

The most direct way to customize how the code generators map XML Schema constructs to Java constructs is to add the customization elements directly to the XML Schema definitions. The JAXB customization elements are placed inside the **`xsd:appinfo`** element of the XML schema construct

that is being modified.

Example 37.3, "Customized XML Schema" shows an example of a schema containing an in-line JAXB customization.

Example 37.3. Customized XML Schema

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <complexType name="size">
    <annotation>
      <appinfo>
        <jaxb:class name="widgetSize" />
      </appinfo>
    </annotation>
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

Using an external binding declaration

When you cannot, or do not want to, make changes to the XML Schema document that defines your type, you can specify the customizations using an external binding declaration. An external binding declaration consists of a number of nested **jaxb:bindings** elements. Example 37.4, "JAXB External Binding Declaration Syntax" shows the syntax of an external binding declaration.

Example 37.4. JAXB External Binding Declaration Syntax

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings [schemaLocation="schemaUri" | wsdlLocation="wsdlUri">
    <jaxb:bindings node="nodeXPath">
      binding declaration
    </jaxb:bindings>
    ...
  </jaxb:bindings>
<jaxb:bindings>
```

The **schemaLocation** attribute and the **wsdlLocation** attribute are used to identify the schema document to which the modifications are applied. Use the **schemaLocation** attribute if you are generating code from a schema document. Use the **wsdlLocation** attribute if you are generating code from a WSDL document.

The **node** attribute is used to identify the specific XML schema construct that is to be modified. It is an XPath statement that resolves to an XML Schema element.

Given the schema document **widgetSchema.xsd**, shown in Example 37.5, "XML Schema File", the external binding declaration shown in Example 37.6, "External Binding Declaration" modifies the generation of the complex type size.

Example 37.5. XML Schema File

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        version="1.0">
  <complexType name="size">
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

Example 37.6. External Binding Declaration

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="wsdlSchema.xsd">
    <jaxb:bindings node="xsd:complexType[@name='size']">
        <jaxb:class name="widgetSize" />
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

To instruct the code generators to use the external binging declaration use the **wsdl2java** tool's **-b *binding-file*** option, as shown below:

```
wsdl2java -b widgetBinding.xml widget.wsdl
```

37.2. SPECIFYING THE JAVA CLASS OF AN XML SCHEMA PRIMITIVE

Overview

By default, XML Schema types are mapped to Java primitive types. While this is the most logical mapping between XML Schema and Java, it does not always meet the requirements of the application developer. You might want to map an XML Schema primitive type to a Java class that can hold extra information, or you might want to map an XML primitive type to a class that allows for simple type substitution.

The JAXB **javaType** customization element allows you to customize the mapping between an XML Schema primitive type and a Java primitive type. It can be used to customize the mappings at both the global level and the individual instance level. You can use the **javaType** element as part of a simple type definition or as part of a complex type definition.

When using the **javaType** customization element you must specify methods for converting the XML representation of the primitive type to and from the target Java class. Some mappings have default conversion methods. For instances where there are no default mappings, Apache CXF provides JAXB methods to ease the development of the required methods.

Syntax

The **javaType** customization element takes four attributes, as described in Table 37.1, "Attributes for Customizing the Generation of a Java Class for an XML Schema Type".

Table 37.1. Attributes for Customizing the Generation of a Java Class for an XML Schema Type

Attribute	Required	Description
name	Yes	Specifies the name of the Java class to which the XML Schema primitive type is mapped. It must be either a valid Java class name or the name of a Java primitive type. You must ensure that this class exists and is accessible to your application. The code generator does not check for this class.
xmlType	No	Specifies the XML Schema primitive type that is being customized. This attribute is only used when the **javaType** element is used as a child of the **globalBindings** element.
parseMethod	No	Specifies the method responsible for parsing the string-based XML representation of the data into an instance of the Java class. For more information see the section called "Specifying the converters".
printMethod	No	Specifies the method responsible for converting a Java object to the string-based XML representation of the data. For more information see the section called "Specifying the converters".

The **javaType** customization element can be used in three ways:

- To modify all instances of an XML Schema primitive type — The **javaType** element modifies all instances of an XML Schema type in the schema document when it is used as a child of the **globalBindings** customization element. When it is used in this manner, you must specify a

value for the **xmlType** attribute that identifies the XML Schema primitive type being modified.

Example 37.7, "Global Primitive Type Customization" shows an in-line global customization that instructs the code generators to use **java.lang.Integer** for all instances of xsd:short in the schema.

Example 37.7. Global Primitive Type Customization

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings ...>
        <jaxb:javaType name="java.lang.Integer"
                       xmlType="xsd:short" />
      </globalBindings
    </appinfo>
  </annotation>
  ...
</schema>
```

- To modify a simple type definition — The **javaType** element modifies the class generated for all instances of an XML simple type when it is applied to a named simple type definition. When using the **javaType** element to modify a simple type definition, do not use the **xmlType** attribute.

  Example 37.8, "Binding File for Customizing a Simple Type" shows an external binding file that modifies the generation of a simple type named zipCode.

  Example 37.8. Binding File for Customizing a Simple Type

  ```
  <jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 jaxb:version="2.0">
    <jaxb:bindings wsdlLocation="widgets.wsdl">
      <jaxb:bindings node="xsd:simpleType[@name='zipCode']">
          <jaxb:javaType
  name="com.widgetVendor.widgetTypes.zipCodeType"

  parseMethod="com.widgetVendor.widgetTypes.support.parseZipCode"

  printMethod="com.widgetVendor.widgetTypes.support.printZipCode" />
      </jaxb:bindings>
    </jaxb:bindings>
  <jaxb:bindings>
  ```

- To modify an element or attribute of a complex type definition — The **javaType** can be applied to individual parts of a complex type definition by including it as part of a JAXB property customization. The **javaType** element is placed as a child to the property's **baseType**

element. When using the **javaType** element to modify a specific part of a complex type definition, do not use the **xmlType** attribute.

Example 37.9, "Binding File for Customizing an Element in a Complex Type" shows a binding file that modifies an element of a complex type.

Example 37.9. Binding File for Customizing an Element in a Complex Type

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings
node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='cost']">
        <jaxb:property>
          <jaxb:baseType>
            <jaxb:javaType
name="com.widgetVendor.widgetTypes.costType"
                           parseMethod="parseCost"
                           printMethod="printCost" >
          </jaxb:baseType>
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

For more information on using the **baseType** element see Section 37.6, "Specifying the Base Type of an Element or an Attribute".

Specifying the converters

The Apache CXF cannot convert XML Schema primitive types into random Java classes. When you use the **javaType** element to customize the mapping of an XML Schema primitive type, the code generator creates an adapter class that is used to marshal and unmarshal the customized XML Schema primitive type. A sample adapter class is shown in Example 37.10, "JAXB Adapter Class".

Example 37.10. JAXB Adapter Class

```
public class Adapter1 extends XmlAdapter<String, javaType>
{
  public javaType unmarshal(String value)
  {
    return(parseMethod(value));
  }

  public String marshal(javaType value)
  {
    return(printMethod(value));
  }
}
```

parseMethod and *printMethod* are replaced by the value of the corresponding **parseMethod** attribute and **printMethod** attribute. The values must identify valid Java methods. You can specify the method's name in one of two ways:

- A fully qualified Java method name in the form of *packagename.ClassName.methodName*
- A simple method name in the form of *methodName*

  When you only provide a simple method name, the code generator assumes that the method exists in the class specified by the **javaType** element's **name** attribute.

IMPORTANT

The code generators **do not** generate parse or print methods. You are responsible for supplying them. For information on developing parse and print methods see the section called “Implementing converters”.

If a value for the **parseMethod** attribute is not provided, the code generator assumes that the Java class specified by the **name** attribute has a constructor whose first parameter is a Java **String** object. The generated adapter's **unmarshal()** method uses the assumed constructor to populate the Java object with the XML data.

If a value for the **printMethod** attribute is not provided, the code generator assumes that the Java class specified by the **name** attribute has a **toString()** method. The generated adapter's **marshal()** method uses the assumed **toString()** method to convert the Java object to XML data.

If the **javaType** element's **name** attribute specifies a Java primitive type, or one of the Java primitive's wrapper types, the code generators use the default converters. For more information on default converters see the section called “Default primitive type converters”.

What is generated

As mentioned in the section called “Specifying the converters”, using the **javaType** customization element triggers the generation of one adapter class for each customization of an XML Schema primitive type. The adapters are named in sequence using the pattern **Adapter*N***. If you specify two primitive type customizations, the code generators create two adapter classes: **Adapter1** and **Adapter2**.

The code generated for an XML schema construct depends on whether the effected XML Schema construct is a globally defined element or is defined as part of a complex type.

When the XML Schema construct is a globally defined element, the object factory method generated for the type is modified from the default method as follows:

- The method is decorated with an **@XmlJavaTypeAdapter** annotation.

  The annotation instructs the runtime which adapter class to use when processing instances of this element. The adapter class is specified as a class object.

- The default type is replaced by the class specified by the **javaType** element's **name** attribute.

Example 37.11, “Customized Object Factory Method for a Global Element” shows the object factory method for an element affected by the customization shown in Example 37.7, “Global Primitive Type Customization”.

■

Example 37.11. Customized Object Factory Method for a Global Element

```
@XmlElementDecl(namespace = "http://widgetVendor.com/types/widgetTypes",
name = "shorty")
    @XmlJavaTypeAdapter(org.w3._2001.xmlschema.Adapter1 .class)
    public JAXBElement<Integer> createShorty(Integer value) {
        return new JAXBElement<Integer>(_Shorty_QNAME, Integer.class,
null, value);
    }
```

When the XML Schema construct is defined as part of a complex type, the generated Java property is modified as follows:

- The property is decorated with an **`@XmlJavaTypeAdapter`** annotation.

  The annotation instructs the runtime which adapter class to use when processing instances of this element. The adapter class is specified as a class object.

- The property's **`@XmlElement`** includes a type property.

  The value of the type property is the class object representing the generated object's default base type. In the case of XML Schema primitive types, the class is **`String`**.

- The property is decorated with an **`@XmlSchemaType`** annotation.

  The annotation identifies the XML Schema primitive type of the construct.

- The default type is replaced by the class specified by the **`javaType`** element's **`name`** attribute.

Example 37.12, “Customized Complex Type” shows the object factory method for an element affected by the customization shown in Example 37.7, “Global Primitive Type Customization” .

Example 37.12. Customized Complex Type

```
public class NumInventory {

    @XmlElement(required = true, type = String.class)
    @XmlJavaTypeAdapter(Adapter1 .class)
    @XmlSchemaType(name = "short")
    protected Integer numLeft;
    @XmlElement(required = true)
    protected String size;

    public Integer getNumLeft() {
        return numLeft;
    }

    public void setNumLeft(Integer value) {
        this.numLeft = value;
    }

    public String getSize() {
        return size;
    }
```

```
    public void setSize(String value) {
        this.size = value;
    }

}
```

Implementing converters

The Apache CXF runtime has does not know how to convert XML primitive types to and from the Java class specified by the **javaType** element, except that it should call the methods specified by the **parseMethod** attribute and the **printMethod** attribute. You are responsible for providing implementations of the methods the runtime calls. The implemented methods must be capable of working with the lexical structures of the XML primitive type.

To simplify the implementation of the data conversion methods, Apache CXF provides the **javax.xml.bind.DatatypeConverter** class. This class provides methods for parsing and printing all of the XML Schema primitive types. The parse methods take string representations of the XML data and they return an instance of the default type defined in Table 33.1, "XML Schema Primitive Type to Java Native Type Mapping". The print methods take an instance of the default type and they return a string representation of the XML data.

The Java documentation for the **DatatypeConverter** class can be found at http://java.sun.com/webservices/docs/1.6/api/javax/xml/bind/DatatypeConverter.html.

Default primitive type converters

When specifying a Java primitive type, or one of the Java primitive type Wrapper classes, in the **javaType** element's **name** attribute, it is not necessary to specify values for the **parseMethod** attribute or the **printMethod** attribute. The Apache CXF runtime substitutes default converters if no values are provided.

The default data converters use the JAXB **DatatypeConverter** class to parse the XML data. The default converters will also provide any type casting necessary to make the conversion work.

37.3. GENERATING JAVA CLASSES FOR SIMPLE TYPES

Overview

By default, named simple types do not result in generated types unless they are enumerations. Elements defined using a simple type are mapped to properties of a Java primitive type.

There are instances when you need to have simple types generated into Java classes, such as is when you want to use type substitution.

To instruct the code generators to generate classes for all globally defined simple types, set the **globalBindings** customization element's **mapSimpleTypeDef** to **true**.

Adding the customization

To instruct the code generators to create Java classes for named simple types add the **globalBinding** element's **mapSimpleTypeDef** attribute and set its value to **true**.

Example 37.13, "in-Line Customization to Force Generation of Java Classes for SimpleTypes" shows an in-line customization that forces the code generator to generate Java classes for named simple types.

Example 37.13. in-Line Customization to Force Generation of Java Classes for SimpleTypes

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings mapSimpleTypeDef="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

Example 37.14, "Binding File to Force Generation of Constants" shows an external binding file that customizes the generation of simple types.

Example 37.14. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings mapSimpleTypeDef="true" />
  <jaxb:bindings>
<jaxb:bindings>
```

IMPORTANT

This customization only affects *named* simple types that are defined in the *global* scope.

Generated classes

The class generated for a simple type has one property called value. The value property is of the Java type defined by the mappings in Section 33.1, "Primitive Types". The generated class has a getter and a setter for the value property.

Example 37.16, "Customized Mapping of a Simple Type" shows the Java class generated for the simple type defined in Example 37.15, "Simple Type for Customized Mapping" .

Example 37.15. Simple Type for Customized Mapping

```
<simpleType name="simpleton">
  <restriction base="xsd:string">
    <maxLength value="10"/>
  </restriction>
</simpleType>
```

Example 37.16. Customized Mapping of a Simple Type

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "simpleton", propOrder = {"value"})
public class Simpleton {

    @XmlValue
    protected String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

}
```

37.4. CUSTOMIZING ENUMERATION MAPPING

Overview

If you want enumerated types that are based on a schema type other than xsd:string, you must instruct the code generator to map it. You can also control the name of the generated enumeration constants.

The customization is done using the **`jaxb:typesafeEnumClass`** element along with one or more **`jaxb:typesafeEnumMember`** elements.

There might also be instances where the default settings for the code generator cannot create valid Java identifiers for all of the members of an enumeration. You can customize how the code generators handle this by using an attribute of the **`globalBindings`** customization.

Member name customizer

If the code generator encounters a naming collision when generating the members of an enumeration or if it cannot create a valid Java identifier for a member of the enumeration, the code generator, by default, generates a warning and does not generate a Java enum type for the enumeration.

You can alter this behavior by adding the **`globalBinding`** element's **`typesafeEnumMemberName`** attribute. The **`typesafeEnumMemberName`** attribute's values are described in Table 37.2, “Values for Customizing Enumeration Member Name Generation”.

Table 37.2. Values for Customizing Enumeration Member Name Generation

Value	Description
`skipGeneration`(default)	Specifies that the Java enum type is not generated and generates a warning.

Value	Description
generateName	Specifies that member names will be generated following the pattern **VALUE_*N***. *N* starts off at one, and is incremented for each member of the enumeration.
generateError	Specifies that the code generator generates an error when it cannot map an enumeration to a Java enum type.

Example 37.17, "Customization to Force Type Safe Member Names" shows an in-line customization that forces the code generator to generate type safe member names.

Example 37.17. Customization to Force Type Safe Member Names

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings typesafeEnumMemberName="generateName" />
    </appinfo>
  </annotation>
  ...
</schema>
```

Class customizer

The **jaxb:typesafeEnumClass** element specifies that an XML Schema enumeration should be mapped to a Java enum type. It has two attributes that are described in Table 37.3, "Attributes for Customizing a Generated Enumeration Class". When the **jaxb:typesafeEnumClass** element is specified in-line, it must be placed inside the **xsd:annotation** element of the simple type it is modifying.

Table 37.3. Attributes for Customizing a Generated Enumeration Class

Attribute	Description
name	Specifies the name of the generated Javaenum type. This value must be a valid Java identifier.
map	Specifies if the enumeration should be mapped to a Java enum type. The default value is**true**.

Member customizer

The **jaxb:typesafeEnumMember** element specifies the mapping between an XML Schema **enumeration** facet and a Java enum type constant. You must use one **jaxb:typesafeEnumMember** element for each **enumeration** facet in the enumeration being customized.

When using in-line customization, this element can be used in one of two ways:

- It can be placed inside the **xsd:annotation** element of the **enumeration** facet it is modifying.
- They can all be placed as children of the **jaxb:typesafeEnumClass** element used to customize the enumeration.

The **jaxb:typesafeEnumMember** element has a **name** attribute that is required. The **name** attribute specifies the name of the generated Java enum type constant. It's value must be a valid Java identifier.

The **jaxb:typesafeEnumMember** element also has a **value** attribute. The **value** is used to associate the **enumeration** facet with the proper **jaxb:typesafeEnumMember** element. The value of the **value** attribute must match one of the values of an **enumeration** facets' **value** attribute. This attribute is required when you use an external binding specification for customizing the type generation, or when you group the **jaxb:typesafeEnumMember** elements as children of the **jaxb:typesafeEnumClass** element.

Examples

Example 37.18, “In-line Customization of an Enumerated Type” shows an enumerated type that uses in-line customization and has the enumeration's members customized separately.

Example 37.18. In-line Customization of an Enumerated Type

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass />
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="one" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="2">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="two" />
          </appinfo>
        </annotation>
      </enumeration>
```

```
      <enumeration value="3">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="three" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="4">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="four" />
          </appinfo>
        </annotation>
      </enumeration>
    </restriction>
  </simpleType>
<schema>
```

Example 37.19, “In-line Customization of an Enumerated Type Using a Combined Mapping” shows an enumerated type that uses in-line customization and combines the member's customization in the class customization.

Example 37.19. In-line Customization of an Enumerated Type Using a Combined Mapping

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass>
            <jaxb:typesafeEnumMember value="1" name="one" />
            <jaxb:typesafeEnumMember value="2" name="two" />
            <jaxb:typesafeEnumMember value="3" name="three" />
            <jaxb:typesafeEnumMember value="4" name="four" />
        </jaxb:typesafeEnumClass>
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1" />
      <enumeration value="2" />
      <enumeration value="3" />
      <enumeration value="4" >
    </restriction>
  </simpleType>
<schema>
```

Example 37.20, “Binding File for Customizing an Enumeration” shows an external binding file that customizes an enumerated type.

Example 37.20. Binding File for Customizing an Enumeration

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:simpleType[@name='widgetInteger']">
        <jaxb:typesafeEnumClass>
            <jaxb:typesafeEnumMember value="1" name="one" />
            <jaxb:typesafeEnumMember value="2" name="two" />
            <jaxb:typesafeEnumMember value="3" name="three" />
            <jaxb:typesafeEnumMember value="4" name="four" />
        </jaxb:typesafeEnumClass>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

37.5. CUSTOMIZING FIXED VALUE ATTRIBUTE MAPPING

Overview

By default, the code generators map attributes defined as having a fixed value to normal properties. When using schema validation, Apache CXF can enforce the schema definition. However, using schema validation increases message processing time.

Another way to map attributes that have fixed values to Java is to map them to Java constants. You can instruct the code generator to map fixed value attributes to Java constants using the **`globalBindings`** customization element. You can also customize the mapping of fixed value attributes to Java constants at a more localized level using the **`property`** element.

Global customization

You can alter this behavior by adding the **`globalBinding`** element's **`fixedAttributeAsConstantProperty`** attribute. Setting this attribute to **`true`** instructs the code generator to map any attribute defined using **`fixed`** attribute to a Java constant.

Example 37.21, “in-Line Customization to Force Generation of Constants” shows an in-line customization that forces the code generator to generate constants for attributes with fixed values.

Example 37.21. in-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
    </appinfo>
```

```
  </annotation>
  ...
</schema>
```

Example 37.22, “Binding File to Force Generation of Constants” shows an external binding file that customizes the generation of fixed attributes.

Example 37.22. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
  <jaxb:bindings>
<jaxb:bindings>
```

Local mapping

You can customize attribute mapping on a per-attribute basis using the **`property`** element's **`fixedAttributeAsConstantProperty`** attribute. Setting this attribute to **`true`** instructs the code generator to map any attribute defined using **`fixed`** attribute to a Java constant.

Example 37.23, “In-Line Customization to Force Generation of Constants” shows an in-line customization that forces the code generator to generate constants for a single attribute with a fixed value.

Example 37.23. In-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <complexType name="widgetAttr">
    <sequence>
      ...
    </sequence>
    <attribute name="fixer" type="xsd:int" fixed="7">
      <annotation>
        <appinfo>
          <jaxb:property fixedAttributeAsConstantProperty="true" />
        </appinfo>
      </annotation>
     </attribute>
  </complexType>
  ...
</schema>
```

Example 37.24, "Binding File to Force Generation of Constants" shows an external binding file that customizes the generation of a fixed attribute.

Example 37.24. Binding File to Force Generation of Constants

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:bindings node="xsd:complexType[@name='widgetAttr']">
      <jaxb:bindings node="xsd:attribute[@name='fixer']">
            <jaxb:property fixedAttributeAsConstantProperty="true" />
        </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

Java mapping

In the default mapping, all attributes are mapped to standard Java properties with getter and setter methods. When this customization is applied to an attribute defined using the **fixed** attribute, the attribute is mapped to a Java constant, as shown in Example 37.25, "Mapping of a Fixed Value Attribute to a Java Constant".

Example 37.25. Mapping of a Fixed Value Attribute to a Java Constant

```
@XmlAttribute
public final static type NAME = value;
```

type is determined by mapping the base type of the attribute to a Java type using the mappings described in Section 33.1, "Primitive Types".

NAME is determined by converting the value of the **attribute** element's **name** attribute to all capital letters.

value is determined by the value of the **attribute** element's **fixed** attribute.

For example, the attribute defined in Example 37.23, "In-Line Customization to Force Generation of Constants" is mapped as shown in Example 37.26, "Fixed Value Attribute Mapped to a Java Constant".

Example 37.26. Fixed Value Attribute Mapped to a Java Constant

```
@XmlRootElement(name = "widgetAttr")
public class WidgetAttr {

    ...

  @XmlAttribute
  public final static int FIXER = 7;
```

```
    ...
}
```

37.6. SPECIFYING THE BASE TYPE OF AN ELEMENT OR AN ATTRIBUTE

Overview

Occasionally you need to customize the class of the object generated for an element, or for an attribute defined as part of an XML Schema complex type. For example, you might want to use a more generalized class of object to allow for simple type substitution.

One way to do this is to use the JAXB base type customization. It allows a developer, on a case by case basis, to specify the class of object generated to represent an element or an attribute. The base type customization allows you to specify an alternate mapping between the XML Schema construct and the generated Java object. This alternate mapping can be a simple specialization or a generalization of the default base class. It can also be a mapping of an XML Schema primitive type to a Java class.

Customization usage

To apply the JAXB base type property to an XML Schema construct use the JAXB **baseType** customization element. The **baseType** customization element is a child of the JAXB **property** element, so it must be properly nested.

Depending on how you want to customize the mapping of the XML Schema construct to Java object, you add either the **baseType** customization element's **name** attribute, or a **javaType** child element. The **name** attribute is used to map the default class of the generated object to another class within the same class hierarchy. The **javaType** element is used when you want to map XML Schema primitive types to a Java class.

IMPORTANT

You cannot use both the **name** attribute and a **javaType** child element in the same **baseType** customization element.

Specializing or generalizing the default mapping

The **baseType** customization element's **name** attribute is used to redefine the class of the generated object to a class within the same Java class hierarchy. The attribute specifies the fully qualified name of the Java class to which the XML Schema construct is mapped. The specified Java class **must** be either a super-class or a sub-class of the Java class that the code generator normally generates for the XML Schema construct. For XML Schema primitive types that map to Java primitive types, the wrapper class is used as the default base class for the purpose of customization.

For example, an element defined as being of xsd:int uses **java.lang.Integer** as its default base class. The value of the **name** attribute can specify any super-class of **Integer** such as **Number** or **Object**.

TIP

For simple type substitution, the most common customization is to map the primitive types to an `Object` object.

Example 37.27, "In-Line Customization of a Base Type" shows an in-line customization that maps one element in a complex type to a Java `Object` object.

Example 37.27. In-Line Customization of a Base Type

```
<complexType name="widgetOrderInfo">
  <all>
    <element name="amount" type="xsd:int" />
     <element name="shippingAdress" type="Address>
      <annotation>
        <appinfo>
           <jaxb:property>
             <jaxb:baseType name="java.lang.Object" />
           </jaxb:property>
        </appinfo>
      </annotation>
    </element>
    <element name="type" type="xsd:string"/>
  </all>
</complexType>
```

Example 37.28, "External Binding File to Customize a Base Type" shows an external binding file for the customization shown in Example 37.27, "In-Line Customization of a Base Type".

Example 37.28. External Binding File to Customize a Base Type

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='shippingAddress']">
        <jaxb:property>
          <jaxb:baseType name="java.lang.Object" />
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

The resulting Java object's `@XmlElement` annotation includes a type property. The value of the type property is the class object representing the generated object's default base type. In the case of XML Schema primitive types, the class is the wrapper class of the corresponding Java primitive type.

Example 37.29, "Java Class with a Modified Base Class" shows the class generated based on the schema definition in Example 37.28, "External Binding File to Customize a Base Type".

Example 37.29. Java Class with a Modified Base Class

```
public class WidgetOrderInfo {

    protected int amount;
    @XmlElement(required = true)
    protected String type;
    @XmlElement(required = true, type = Address.class)
    protected Object shippingAddress;

    ...
    public Object getShippingAddress() {
        return shippingAddress;
    }


    public void setShippingAddress(Object value) {
        this.shippingAddress = value;
    }

}
```

Usage with `javaType`

The **`javaType`** element can be used to customize how elements and attributes defined using XML Schema primitive types are mapped to Java objects. Using the **`javaType`** element provides a lot more flexibility than simply using the **`baseType`** element's **`name`** attribute. The **`javaType`** element allows you to map a primitive type to any class of object.

For a detailed description of using the **`javaType`** element, see Section 37.2, “Specifying the Java Class of an XML Schema Primitive”.

CHAPTER 38. USING A JAXBCONTEXT OBJECT

Abstract

The **JAXBContext** object allows the Apache CXF's runtime to transform data between XML elements and Java object. Application developers need to instantiate a **JAXBContext** object they want to use JAXB objects in message handlers and when implementing consumers that work with raw XML messages.

OVERVIEW

The **JAXBContext** object is a low-level object used by the runtime. It allows the runtime to convert between XML elements and their corresponding Java representations. An application developer generally does not need to work with **JAXBContext** objects. The marshaling and unmarshaling of XML data is typically handled by the transport and binding layers of a JAX-WS application.

However, there are instances when an application will need to manipulate the XML message content directly. In two of these instances:

- Implementing consumers that use raw XML data
- Working with messages in a handler

You will need instantiate a **JAXBContext** object using one of the two available **JAXBContext.newInstance()** methods.

BEST PRACTICES

JAXBContext objects are resource intensive to instantiate. It is recommended that an application create as few instances as possible. One way to do this is to create a single **JAXBContext** object that can manage all of the JAXB objects used by your application and share it among as many parts of your application as possible.

TIP

JAXBContext objects are thread safe.

GETTING A JAXBCONTEXT OBJECT USING AN OBJECT FACTORY

The **JAXBContext** class provides a **newInstance()** method, shown in Example 38.1, "Getting a JAXB Context Using Classes", that takes a list of classes that implement JAXB objects.

Example 38.1. Getting a JAXB Context Using Classes

```
static JAXBContext newInstance(Class... classesToBeBound)
    throws JAXBException;
```

The returned **JAXBObject** object will be able to marshal and unmarshal data for the JAXB object implemented by the classes passed into the method. It will also be able to work with any classes that are statically referenced from any of the classes passed into the method.

While it is possible to pass the name of every JAXB class used by your application to the **newInstance()** method it is not efficient. A more efficient way to accomplish the same goal is to pass in the object factory, or object factories, generated for your application. The resulting **JAXBContext** object will be able to manage any JAXB classes the specified object factories can instantiate.

GETTING A JAXBCONTEXT OBJECT USING PACKAGE NAMES

The **JAXBContext** class provides a **newInstance()** method, shown in Example 38.2, "Getting a JAXB Context Using Classes", that takes a colon (**:**) seperated list of package names. The specified packages should contain JAXB objects derived from XML Schema.

Example 38.2. Getting a JAXB Context Using Classes

```
static JAXBContext newInstance(String contextPath)
    throws JAXBException;
```

The returned **JAXBContext** object will be able to marshal and unmarshal data for all of the JAXB objects implemented by the classes in the specified packages.

CHAPTER 39. USING SOAP OVER JMS

OVERVIEW

The SOAP over JMS protocol is defined by the World Wide Web Consortium(W3C) as a way of providing a more reliable transport layer to the customary SOAP/HTTP protocol used by most services. The Apache CXF implementation is fully compliant with the specification and should be compatible with any framework that is also compliant.

This transport uses JNDI to find the JMS destinations. When an operation is invoked, the request is packaged as a SOAP message and sent in the body of a JMS message to the specified destination.

Publishing and consuming SOAP/JMS services differ from SOAP/HTTP services in the following ways:

- SOAP/JMS service addressed are specified using a special JMS URI
- you must use the Apache CXF specific factory objects to use SOAP/JMS endpoints

JMS URIS

JMS endpoints use a JMS URI as defined in the URI Scheme for JMS 1.0 . Example 39.1, "JMS URI syntax" shows the syntax for a JMS URI.

Example 39.1. JMS URI syntax

```
jms:variant:destination?options
```

Table 39.1, "JMS URI variants" describes the available variants for the JMS URI.

Table 39.1. JMS URI variants

Variant	Description
jndi	Specifies that the destination is a JNDI name for the target destination. When using this variant, you must provide the configuration for accessing the JNDI provider.
topic	Specifies that the destination is the name of the topic to be used as the target destination. The string provided is passed into `Session.createTopic()` to create a representation of the destination.
queue	Specifies that the destination is the name of the queue to be used as the target destination. The string provided is passed into `Session.createQueue()` to create a representation of the destination.

Table 39.2, "JMS properties settable as URI options" shows the URI options.

Table 39.2. JMS properties settable as URI options

Property	Default	Description
`deliveryMode`	`PERSISTENT`	Specifies whether to use JMS `PERSISTENT` or `NON_PERSISTENT` message semantics. In the case of `PERSISTENT` delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas `NON_PERSISTENT` messages are kept in memory only.
`replyToName`		Explicitly specifies the reply destination to appear in the `JMSReplyTo` header. Setting this property is recommended for applications that have request-reply semantics because the JMS provider will assign a temporary reply queue if one is not explicitly set. The value of this property has an interpretation that depends on the variant specified in the JMS URI: • `jndi` variant—the JNDI name of the destination • `queue` or `topic` variants—the actual name of the destination
`priority`	`4`	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`timeToLive`	`0`	Time (in milliseconds) after which the message will be discarded by the JMS provider. `0` represents an infinite lifetime.
`jndiConnectionFactoryName`		Specifies the JNDI name of the JMS connection factory.

Property	Default	Description
jndiInitialContextFactory		Specifies the fully qualified Java class name of the JNDI provider (which must be of **javax.jms.InitialContextFactory** type). Equivalent to setting the **java.naming.factory.initial** Java system property.
jndiURL		Specifies the URL that initializes the JNDI provider. Equivalent to setting the **java.naming.provider.url** Java system property.

PUBLISHING A SERVICE

The JAX-WS standard **publish()** method cannot be used to publish a SOAP/JMS service. Instead, you must use the Apache CXF's **JaxWsServerFactoryBean** class as shown in Example 39.2, "Publishing a SOAP/JMS service".

Example 39.2. Publishing a SOAP/JMS service

```
1 String address =
   "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
    + "?jndiInitialContextFactory"
    + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
    + "&jndiConnectionFactoryName=ConnectionFactory"
    + "&jndiURL=tcp://localhost:61500";
Hello implementor = new HelloImpl();
2 JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();
  svrFactory.setServiceClass(Hello.class);
3 svrFactory.setAddress(address);
  4
svrFactory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANS
PORTID);
svrFactory.setServiceBean(implementor);
svrFactory.create();
```

The code in Example 39.2, "Publishing a SOAP/JMS service" does the following:

1. Creates the JMS URI representing t he endpoint's address.
2. Instantiates a **JaxWsServerFactoryBean** to publish the service.
3. Sets the **address** field of the factory bean with the JMS URI of the service.
4. Specifies that the service created by the factory will use the SOAP/JMS transport.

CONSUMING A SERVICE

The standard JAX-WS APIs cannot be used to consume a SOAP/JMS service. Instead, you must use the Apache CXF's **JaxWsProxyFactoryBean** class as shown in Example 39.3, "Consuming a SOAP/JMS service".

Example 39.3. Consuming a SOAP/JMS service

```
// Java
public void invoke() throws Exception {
(1)    String address =
  "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
        + "?jndiInitialContextFactory"
        + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
        +
"&jndiConnectionFactoryName=ConnectionFactory&jndiURL=tcp://localhost:61
500";
(2)    JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
(3)     factory.setAddress(address);
(4)
factory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANSPOR
TID);
    factory.setServiceClass(Hello.class);
    Hello client = (Hello)factory.create();
    String reply = client.sayHi(" HI");
    System.out.println(reply);
}
```

The code in Example 39.3, "Consuming a SOAP/JMS service" does the following:

1. Creates the JMS URI representing t he endpoint's address.
2. Instantiates a **JaxWsProxyFactoryBean** to create the proxy.
3. Sets the **address** field of the factory bean with the JMS URI of the service.
4. Specifies that the proxy created by the factory will use the SOAP/JMS transport.

CHAPTER 40. DEVELOPING ASYNCHRONOUS APPLICATIONS

Abstract

JAX-WS provides an easy mechanism for accessing services asynchronously. The SEI can specify additional methods that can be used to access a service asynchronously. The Apache CXF code generators generate the extra methods for you. You simply add the business logic.

In addition to the usual synchronous mode of invocation, Apache CXF supports two forms of asynchronous invocation:

- Polling approach — To invoke the remote operation using the polling approach, you call a method that has no output parameters, but returns a **`javax.xml.ws.Response`** object. The **`Response`** object (which inherits from the **`javax.util.concurrency.Future`** interface) can be polled to check whether or not a response message has arrived.
- Callback approach — To invoke the remote operation using the callback approach, you call a method that takes a reference to a callback object (of **`javax.xml.ws.AsyncHandler`** type) as one of its parameters. When the response message arrives at the client, the runtime calls back on the **`AsyncHandler`** object, and gives it the contents of the response message.

40.1. WSDL FOR ASYNCHRONOUS EXAMPLES

Example 40.1, “WSDL Contract for Asynchronous Example” shows the WSDL contract that is used for the asynchronous examples. The contract defines a single interface, **`GreeterAsync`**, which contains a single operation, greetMeSometime.

Example 40.1. WSDL Contract for Asynchronous Example

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions
xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:tns="http://apache.org/hello_world_async_soap_http"

xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://apache.org/hello_world_async_soap_http"
                  name="HelloWorld">
  <wsdl:types>
    <schema
targetNamespace="http://apache.org/hello_world_async_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"

xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
            elementFormDefault="qualified">
      <element name="greetMeSometime">
        <complexType>
          <sequence>
            <element name="requestType" type="xsd:string"/>
          </sequence>
        </complexType>
```

```
      </element>
      <element name="greetMeSometimeResponse">
        <complexType>
          <sequence>
            <element name="responseType"
                     type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>

  <wsdl:message name="greetMeSometimeRequest">
    <wsdl:part name="in" element="x1:greetMeSometime"/>
  </wsdl:message>
  <wsdl:message name="greetMeSometimeResponse">
    <wsdl:part name="out"
               element="x1:greetMeSometimeResponse"/>
  </wsdl:message>

  <wsdl:portType name="GreeterAsync">
    <wsdl:operation name="greetMeSometime">
      <wsdl:input name="greetMeSometimeRequest"
                  message="tns:greetMeSometimeRequest"/>
      <wsdl:output name="greetMeSometimeResponse"
                   message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="GreeterAsync_SOAPBinding"
                type="tns:GreeterAsync">
    ...
  </wsdl:binding>

  <wsdl:service name="SOAPService">
    <wsdl:port name="SoapPort"
               binding="tns:GreeterAsync_SOAPBinding">
      <soap:address
location="http://localhost:9000/SoapContext/SoapPort"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

40.2. GENERATING THE STUB CODE

Overview

The asynchronous style of invocation requires extra stub code for the dedicated asynchronous methods defined on the SEI. This special stub code is not generated by default. To switch on the asynchronous feature and generate the requisite stub code, you must use the mapping customization feature from the WSDL 2.0 specification.

Customization enables you to modify the way the Maven code generation plug-in generates stub code. In particular, it enables you to modify the WSDL-to-Java mapping and to switch on certain features.

Here, customization is used to switch on the asynchronous invocation feature. Customizations are specified using a binding declaration, which you define using a **jaxws:bindings** tag (where the **jaxws** prefix is tied to the **http://java.sun.com/xml/ns/jaxws** namespace). There are two ways of specifying a binding declaration:

External Binding Declaration

When using an external binding declaration the **jaxws:bindings** element is defined in a file separate from the WSDL contract. You specify the location of the binding declaration file to code generator when you generate the stub code.

Embedded Binding Declaration

When using an embedded binding declaration you embed the **jaxws:bindings** element directly in a WSDL contract, treating it as a WSDL extension. In this case, the settings in **jaxws:bindings** apply only to the immediate parent element.

Using an external binding declaration

The template for a binding declaration file that switches on asynchronous invocations is shown in Example 40.2, “Template for an Asynchronous Binding Declaration” .

Example 40.2. Template for an Asynchronous Binding Declaration

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          wsdlLocation="AffectedWSDL"
          xmlns="http://java.sun.com/xml/ns/jaxws">
  <bindings node="AffectedNode">
    <enableAsyncMapping>true</enableAsyncMapping>
  </bindings>
</bindings>
```

Where *AffectedWSDL* specifies the URL of the WSDL contract that is affected by this binding declaration. The *AffectedNode* is an XPath value that specifies which node (or nodes) from the WSDL contract are affected by this binding declaration. You can set *AffectedNode* to **wsdl:definitions**, if you want the entire WSDL contract to be affected. The **jaxws:enableAsyncMapping** element is set to **true** to enable the asynchronous invocation feature.

For example, if you want to generate asynchronous methods only for the **GreeterAsync** interface, you can specify <bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']"> in the preceding binding declaration.

Assuming that the binding declaration is stored in a file, **async_binding.xml**, you would set up your POM as shown in Example 40.3, “Consumer Code Generation” .

Example 40.3. Consumer Code Generation

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
```

```
      <execution>
        <id>generate-sources</id>
        <phase>generate-sources</phase>
        <configuration>
          <sourceRoot>outputDir</sourceRoot>
          <wsdlOptions>
            <wsdlOption>
              <wsdl>hello_world.wsdl</wsdl>
              <extraargs>
                <extraarg>-client</extraarg>
                <extraarg>-b async_binding.xml</extraarg>
              </extraargs>
            </wsdlOption>
          </wsdlOptions>
        </configuration>
        <goals>
          <goal>wsdl2java</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

The **-b** option tells the code generator where to locate the external binding file.

For more information on the code generator see cxf-codegen-plugin.

Using an embedded binding declaration

You can also embed the binding customization directly into the WSDL document defining the service by placing the **jaxws:bindings** element and its associated **jaxws:enableAsynchMapping** child directly into the WSDL. You also must add a namespace declaration for the **jaxws** prefix.

Example 40.4, "WSDL with Embedded Binding Declaration for Asynchronous Mapping" shows a WSDL file with an embedded binding declaration that activates the asynchronous mapping for an operation.

Example 40.4. WSDL with Embedded Binding Declaration for Asynchronous Mapping

```
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
                  ...
                  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
                  ...>
  ...
  <wsdl:portType name="GreeterAsync">
    <wsdl:operation name="greetMeSometime">
      <jaxws:bindings>
        <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
      </jaxws:bindings>
      <wsdl:input name="greetMeSometimeRequest"
                  message="tns:greetMeSometimeRequest"/>
      <wsdl:output name="greetMeSometimeResponse"
                   message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
```

```
  </wsdl:portType>
  ...
</wsdl:definitions>
```

When embedding the binding declaration into the WSDL document you can control the scope affected by the declaration by changing where you place the declaration. When the declaration is placed as a child of the **wsdl:definitions** element the code generator creates asynchronous methods for all of the operations defined in the WSDL document. If it is placed as a child of a **wsdl:portType** element the code generator creates asynchronous methods for all of the operations defined in the interface. If it is placed as a child of a **wsdl:operation** element the code generator creates asynchronous methods for only that operation.

It is not necessary to pass any special options to the code generator when using embedded declarations. The code generator will recognize them and act accordingly.

Generated interface

After generating the stub code in this way, the **GreeterAsync** SEI (in the file **GreeterAsync.java**) is defined as shown in Example 40.5, "Service Endpoint Interface with Methods for Asynchronous Invocations".

Example 40.5. Service Endpoint Interface with Methods for Asynchronous Invocations

```
package org.apache.hello_world_async_soap_http;

import
org.apache.hello_world_async_soap_http.types.GreetMeSometimeResponse;
...

public interface GreeterAsync
{
  public Future<?> greetMeSometimeAsync(
        java.lang.String requestType,
        AsyncHandler<GreetMeSometimeResponse> asyncHandler
    );

    public Response<GreetMeSometimeResponse> greetMeSometimeAsync(
        java.lang.String requestType
    );

    public java.lang.String greetMeSometime(
        java.lang.String requestType
    );
}
```

In addition to the usual synchronous method, **greetMeSometime()**, two asynchronous methods are also generated for the greetMeSometime operation:

- Callback approach

```
public Future<?> greetMeSomtimeAsync(java.lang.String requestType,
                                     AsyncHandler<GreetMeSomtimeRespons
e> asyncHandler);
```

- Polling approach

```
public Response<GreetMeSomeTimeResponse> greetMeSometimeAsync(java.lang
.String requestType);
```

40.3. IMPLEMENTING AN ASYNCHRONOUS CLIENT WITH THE POLLING APPROACH

The polling approach is the more straightforward of the two approaches to developing an asynchronous application. The client invokes the asynchronous method called ***OperationName*Async()** and is returned a **Response<T>** object that it polls for a response. What the client does while it is waiting for a response is depends on the requirements of the application. There are two basic patterns for handling the polling:

- Non-blocking polling — You periodically check to see if the result is ready by calling the non-blocking **Response<T>.isDone()** method. If the result is ready, the client processes it. If it not, the client continues doing other things.
- Blocking polling — You call **Response<T>.get()** right away, and block until the response arrives (optionally specifying a timeout).

Using the non-blocking pattern

Example 40.6, "Non-Blocking Polling Approach for an Asynchronous Operation Call" illustrates using non-blocking polling to make an asynchronous invocation on the greetMeSometime operation defined in Example 40.1, "WSDL Contract for Asynchronous Example". The client invokes the asynchronous operation and periodically checks to see if the result is returned.

Example 40.6. Non-Blocking Polling Approach for an Asynchronous Operation Call

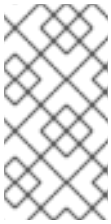
```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {

    // set up the proxy for the client
```

```
1    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
       port.greetMeSometimeAsync(System.getProperty("user.name"));

2      while (!greetMeSomeTimeResp.isDone()) {
       // client does some work
     }
3      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
       // process the response

     System.exit(0);
  }
}
```

The code in Example 40.6, "Non-Blocking Polling Approach for an Asynchronous Operation Call" does the following:

1. Invokes the **`greetMeSometimeAsync()`** on the proxy.

   The method call returns the **`Response<GreetMeSometimeResponse>`** object to the client immediately. The Apache CXF runtime handles the details of receiving the reply from the remote endpoint and populating the **`Response<GreetMeSometimeResponse>`** object.

   NOTE

   The runtime transmits the request to the remote endpoint's **`greetMeSometime()`** method and handles the details of the asynchronous nature of the call transparently. The endpoint, and therefore the service implementation, never worries about the details of how the client intends to wait for a response.

2. Checks to see if a response has arrived by checking the **`isDone()`** of the returned **`Response`** object.

   If the response has not arrived, the client continues working before checking again.

3. When the response arrives, the client retrieves it from the **`Response`** object using the **`get()`** method.

Using the blocking pattern

When using the block polling pattern, the **`Response`** object's **`isDone()`** is never called. Instead, the **`Response`** object's **`get()`** method is called immediately after invoking the remote operation. The **`get()`** blocks until the response is available.

TIP

You can also pass a timeout limit to the **`get()`** method.

Example 40.7, "Blocking Polling Approach for an Asynchronous Operation Call" shows a client that uses blocking polling.

Example 40.7. Blocking Polling Approach for an Asynchronous Operation Call

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {

    // set up the proxy for the client

    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));
    GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
      System.exit(0);
  }
}
```

40.4. IMPLEMENTING AN ASYNCHRONOUS CLIENT WITH THE CALLBACK APPROACH

An alternative approach to making an asynchronous operation invocation is to implement a callback class. You then call the asynchronous remote method that takes the callback object as a parameter. The runtime returns the response to the callback object.

To implement an application that uses callbacks, do the following:

1. Create a callback class that implements the **`AsyncHandler`** interface.

NOTE

Your callback object can perform any amount of response processing required by your application.

2. Make remote invocations using the ***`operationName`*`Async()`** that takes the callback object as a parameter and returns a **`Future<?>`** object.

3. If your client requires access to the response data, you can poll the returned **`Future<?>`** object's **`isDone()`** method to see if the remote endpoint has sent the response.

TIP

If the callback object does all of the response processing, it is not necessary to check if the response has arrived.

Implementing the callback

The callback class must implement the **`javax.xml.ws.AsyncHandler`** interface. The interface defines a single method:

`void handleResponse(Response<T> res);`
The Apache CXF runtime calls the **`handleResponse()`** method to notify the client that the response has arrived. Example 40.8, "The **`javax.xml.ws.AsyncHandler`** Interface" shows an outline of the **`AsyncHandler`** interface that you must implement.

Example 40.8. The `javax.xml.ws.AsyncHandler` Interface

```
public interface javax.xml.ws.AsyncHandler
{
  void handleResponse(Response<T> res)
}
```

Example 40.9, "Callback Implementation Class" shows a callback class for the greetMeSometime operation defined in Example 40.1, "WSDL Contract for Asynchronous Example" .

Example 40.9. Callback Implementation Class

```
package demo.hw.client;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.types.*;

public class GreeterAsyncHandler implements
AsyncHandler<GreetMeSometimeResponse>
{
(1)  private GreetMeSometimeResponse reply;

(2)  public void handleResponse(Response<GreetMeSometimeResponse>
                                response)
  {
    try
    {
      reply = response.get();
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
    }
  }

(3)  public String getResponse()
```

```
    {
      return reply.getResponseType();
    }
  }
```

The callback implementation shown in Example 40.9, "Callback Implementation Class" does the following:

1. Defines a member variable, **response**, that holds the response returned from the remote endpoint.
2. Implements **handleResponse()**.

   This implementation simply extracts the response and assigns it to the member variable **reply**.
3. Implements an added method called **getResponse()**.

   This method is a convenience method that extracts the data from **reply** and returns it.

Implementing the consumer

Example 40.10, "Callback Approach for an Asynchronous Operation Call" illustrates a client that uses the callback approach to make an asynchronous call to the GreetMeSometime operation defined in Example 40.1, "WSDL Contract for Asynchronous Example".

Example 40.10. Callback Approach for an Asynchronous Operation Call

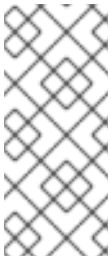
```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  ...

  public static void main(String args[]) throws Exception
  {
    ...
    // Callback approach
1   GreeterAsyncHandler callback = new GreeterAsyncHandler();

2   Future<?> response =
      port.greetMeSometimeAsync(System.getProperty("user.name"),
                                callback);
3   while (!response.isDone())
    {
    // Do some work
  }
```

```
4       resp = callback.getResponse();
        ...
    System.exit(0);
  }
}
```

The code in Example 40.10, "Callback Approach for an Asynchronous Operation Call" does the following:

1. Instantiates a callback object.
2. Invokes the **greetMeSometimeAsync()** that takes the callback object on the proxy.

   The method call returns the **Future<?>** object to the client immediately. The Apache CXF runtime handles the details of receiving the reply from the remote endpoint, invoking the callback object's **handleResponse()** method, and populating the **Response<GreetMeSometimeResponse>** object.

   NOTE

   The runtime transmits the request to the remote endpoint's **greetMeSometime()** method and handles the details of the asynchronous nature of the call without the remote endpoint's knowledge. The endpoint, and therefore the service implementation, does not need to worry about the details of how the client intends to wait for a response.

3. Uses the returned **Future<?>** object's **isDone()** method to check if the response has arrived from the remote endpoint.
4. Invokes the callback object's **getResponse()** method to get the response data.

40.5. CATCHING EXCEPTIONS RETURNED FROM A REMOTE SERVICE

Overview

Consumers making asynchronous requests will not receive the same exceptions returned than when they make synchronous requests. Any exceptions returned to the consumer asynchronously are wrapped in an **ExecutionException** exception. The actual exception thrown by the service is stored in the **ExecutionException** exception's **cause** field.

Catching the exception

Exceptions generated by a remote service are thrown locally by the method that passes the response to the consumer's business logic. When the consumer makes a synchronous request, the method making the remote invocation throws the exception. When the consumer makes an asynchronous request, the **Response<T>** object's **get()** method throws the exception. The consumer will not discover that an error was encountered in processing the request until it attempts to retrieve the response message.

Unlike the methods generated by the JAX-WS framework, the **Response<T>** object's **get()** method does not throw either user modeled exceptions nor the generic JAX-WS exceptions. Instead, it throws a **java.util.concurrent.ExecutionException** exception.

Getting the exception details

The framework stores the exception returned from the remote service in the **ExecutionException** exception's **cause** field. The details about the remote exception are extracted by getting the value of the **cause** field and examining the stored exception. The stored exception can be any user defined exception or one of the generic JAX-WS exceptions.

Example

Example 40.11, “Catching an Exception using the Polling Approach” shows an example of catching an exception using the polling approach.

Example 40.11. Catching an Exception using the Polling Approach

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client
{
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception
  {
    ...
    // port is a previously established proxy object.
    Response<GreetMeSometimeResponse> resp =
       port.greetMeSometimeAsync(System.getProperty("user.name"));

    while (!resp.isDone())
    {
      // client does some work
    }

1   try
    {
    GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
    // process the response
    }
2   catch (ExecutionException ee)
    {
```

```
3         Throwable cause = ee.getCause();
          System.out.println("Exception "+cause.getClass().getName()+"
thrown by the remote service.");
      }
    }
}
```

The code in Example 40.11, “Catching an Exception using the Polling Approach” does the following:

1. Wraps the call to the **Response<T>** object's **get()** method in a try/catch block.
2. Catches a **ExecutionException** exception.
3. Extracts the **cause** field from the exception.

If the consumer was using the callback approach the code used to catch the exception would be placed in the callback object where the service's response is extracted.

CHAPTER 41. USING RAW XML MESSAGES

Abstract

The high-level JAX-WS APIs shield the developer from using native XML messages by marshaling the data into JAXB objects. However, there are cases when it is better to have direct access to the raw XML message data that is passing on the wire. The JAX-WS APIs provide two interfaces that provide access to the raw XML: the **`Dispatch`** interface is the client-side interface, and the **`Provider`** interface is the server-side interface.

41.1. USING XML IN A CONSUMER

The **`Dispatch`** interface is a low-level JAX-WS API that allows you work directly with raw messages. It accepts and returns messages, or payloads, of a number of types including DOM objects, SOAP messages, and JAXB objects. Because it is a low-level API, the **`Dispatch`** interface does not perform any of the message preparation that the higher-level JAX-WS APIs perform. You must ensure that the messages, or payloads, that you pass to the **`Dispatch`** object are properly constructed, and make sense for the remote operation being invoked.

41.1.1. Usage Modes

Overview

`Dispatch` objects have two *usage modes*:

- Message mode
- Message Payload mode (Payload mode)

The usage mode you specify for a **`Dispatch`** object determines the amount of detail that is passed to the user level code.

Message mode

In *message mode*, a **`Dispatch`** object works with complete messages. A complete message includes any binding specific headers and wrappers. For example, a consumer interacting with a service that requires SOAP messages must provide the **`Dispatch`** object's **`invoke()`** method a fully specified SOAP message. The **`invoke()`** method also returns a fully specified SOAP message. The consumer code is responsible for completing and reading the SOAP message's headers and the SOAP message's envelope information.

TIP

Message mode is not ideal when working with JAXB objects.

To specify that a **`Dispatch`** object uses message mode provide the value **`java.xml.ws.Service.Mode.MESSAGE`** when creating the **`Dispatch`** object. For more information about creating a **`Dispatch`** object see the section called “Creating a **`Dispatch`** object”.

Payload mode

In *payload mode*, also called message payload mode, a **Dispatch** object works with only the payload of a message. For example, a **Dispatch** object working in payload mode works only with the body of a SOAP message. The binding layer processes any binding level wrappers and headers. When a result is returned from the **invoke()** method the binding level wrappers and headers are already striped away, and only the body of the message is left.

TIP

When working with a binding that does not use special wrappers, such as the Apache CXF XML binding, payload mode and message mode provide the same results.

To specify that a **Dispatch** object uses payload mode provide the value **java.xml.ws.Service.Mode.PAYLOAD** when creating the **Dispatch** object. For more information about creating a **Dispatch** object see the section called “Creating a Dispatch object”.

41.1.2. Data Types

Overview

Because **Dispatch** objects are low-level objects, they are not optimized for using the same JAXB generated types as the higher level consumer APIs. **Dispatch** objects work with the following types of objects:

- **javax.xml.transform.Source**
- **javax.xml.soap.SOAPMessage**
- **javax.activation.DataSource**
- JAXB

Using Source objects

A **Dispatch** object accepts and returns objects that are derived from the **javax.xml.transform.Source** interface. **Source** objects are supported by any binding, and in either message mode or payload mode.

Source objects are low level objects that hold XML documents. Each **Source** implementation provides methods that access the stored XML documents and then manipulate its contents. The following objects implement the **Source** interface:

DOMSource

Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of **Node** objects that are accessed using the **getNode()** method. Nodes can be either updated or added to the DOM tree using the **setNode()** method.

SAXSource

Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an **InputSource** object that holds the raw data and an **XMLReader** object that parses the raw data.

StreamSource

Holds XML messages as a data stream. The data stream can be manipulated the same as any other data stream.

If you create your **Dispatch** object so that it uses generic **Source** objects, Apache CXF returns the messages as **SAXSource** objects.

This behavior can be changed using the endpoint's source-preferred-format property. See Part IV, "Configuring Web Service Endpoints" for information about configuring the Apache CXF runtime.

Using SOAPMessage objects

Dispatch objects can use **javax.xml.soap.SOAPMessage** objects when the following conditions are true:

- The **Dispatch** object is using the SOAP binding
- The **Dispatch** object is using message mode

A **SOAPMessage** object holds a SOAP message. They contain one **SOAPPart** object and zero or more **AttachmentPart** objects. The **SOAPPart** object contains the SOAP specific portions of the SOAP message including the SOAP envelope, any SOAP headers, and the SOAP message body. The **AttachmentPart** objects contain binary data that is passed as an attachment.

Using DataSource objects

Dispatch objects can use objects that implement the **javax.activation.DataSource** interface when the following conditions are true:

- The **Dispatch** object is using the HTTP binding
- The **Dispatch** object is using message mode

DataSource objects provide a mechanism for working with MIME typed data from a variety of sources, including URLs, files, and byte arrays.

Using JAXB objects

While **Dispatch** objects are intended to be low level APIs that allow you to work with raw messages, they also allow you to work with JAXB objects. To work with JAXB objects a **Dispatch** object must be passed a **JAXBContext** that can marshal and unmarshal the JAXB objects in use. The **JAXBContext** is passed when the **Dispatch** object is created.

You can pass any JAXB object understood by the **JAXBContext** object as the parameter to the **invoke()** method. You can also cast the returned message into any JAXB object understood by the **JAXBContext** object.

For information on creating a **JAXBContext** object see Chapter 38, *Using A **JAXBContext** Object*.

41.1.3. Working with Dispatch Objects

Procedure

To use a **Dispatch** object to invoke a remote service the following sequence should be followed:

1. Create a **Dispatch** object.
2. Construct a request message.
3. Call the proper **invoke()** method.
4. Parse the response message.

Creating a Dispatch object

To create a **Dispatch** object do the following:

1. Create a **Service** object to represent the **wsdl:service** element that defines the service on which the **Dispatch** object will make invocations. See Section 24.1, "Creating a Service Object".
2. Create the **Dispatch** object using the **Service** object's **createDispatch()** method, shown in Example 41.1, "The createDispatch() Method".

Example 41.1. The createDispatch() Method

```
public Dispatch<T> createDispatch(QName portName,
                                  java.lang.Class<T> type,
                                  Service.Mode mode)
    throws WebServiceException;
```

NOTE

If you are using JAXB objects the method signature for **createDispatch()** is:

```
public Dispatch<T> createDispatch(QName portName,
                                  javax.xml.bind.JAXBContext
 context,
                                  Service.Mode mode)
    throws WebServiceException;
```

Table 41.1, "Parameters for createDispatch()" describes the parameters for the **createDispatch()** method.

Table 41.1. Parameters for createDispatch()

Parameter	Description
portName	Specifies the QName of the **wsdl:port** element that represents the service provider where the **Dispatch** object will make invocations.

Parameter	Description
type	Specifies the data type of the objects used by the **Dispatch** object. See Section 41.1.2, "Data Types". When working with JAXB objects, this parameter specifies the **JAXBContext** object used to marshal and unmarshal the JAXB objects.
mode	Specifies the usage mode for the **Dispatch** object. See Section 41.1.1, "Usage Modes".

Example 41.2, "Creating a **Dispatch** Object" shows the code for creating a **Dispatch** object that works with **DOMSource** objects in payload mode.

Example 41.2. Creating a Dispatch Object

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    QName serviceName = new QName("http://org.apache.cxf",
"stockQuoteReporter");
    Service s = Service.create(serviceName);

    QName portName = new QName("http://org.apache.cxf",
"stockQuoteReporterPort");
    Dispatch<DOMSource> dispatch = s.createDispatch(portName,
                                                    DOMSource.class,

Service.Mode.PAYLOAD);
    ...
```

Constructing request messages

When working with **Dispatch** objects, requests must be built from scratch. The developer is responsible for ensuring that the messages passed to a **Dispatch** object match a request that the targeted service provider can process. This requires precise knowledge about the messages used by the service provider and what, if any, header information it requires.

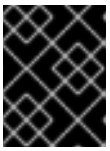
This information can be provided by a WSDL document or an XML Schema document that defines the messages. While service providers vary greatly there are a few guidelines to be followed:

- The root element of the request is based in the value of the **name** attribute of the **wsdl:operation** element corresponding to the operation being invoked.

WARNING

If the service being invoked uses doc/literal bare messages, the root element of the request is based on the value of the **name** attribute of the **wsdl:part** element referred to by the **wsdl:operation** element.

- The root element of the request is namespace qualified.
- If the service being invoked uses rpc/literal messages, the top-level elements in the request will not be namespace qualified.



IMPORTANT

The children of top-level elements may be namespace qualified. To be certain you must check their schema definitions.

- If the service being invoked uses rpc/literal messages, none of the top-level elements can be null.
- If the service being invoked uses doc/literal messages, the schema definition of the message determines if any of the elements are namespace qualified.

For more information about how services use XML messages see, the WS-I Basic Profile.

Synchronous invocation

For consumers that make synchronous invocations that generate a response, use the **Dispatch** object's **invoke()** method shown in Example 41.3, "The **Dispatch.invoke()** Method".

Example 41.3. The Dispatch.invoke() Method

```
T invoke(T msg)
    throws WebServiceException;
```

The type of both the response and the request passed to the **invoke()** method are determined when the **Dispatch** object is created. For example if you create a **Dispatch** object using **createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE)**, both the response and the request are **SOAPMessage** objects.

NOTE

When using JAXB objects, both the response and the request can be of any type the provided **JAXBContext** object can marshal and unmarshal. Also, the response and the request can be different JAXB objects.

Example 41.4, "Making a Synchronous Invocation Using a `Dispatch` Object" shows code for making a synchronous invocation on a remote service using a `DOMSource` object.

Example 41.4. Making a Synchronous Invocation Using a `Dispatch` Object

```
// Creating a DOMSource Object for the request
DocumentBuilder db = DocumentBuilderFactory.newDocumentBuilder();
Document requestDoc = db.newDocument();
Element root =
requestDoc.createElementNS("http://org.apache.cxf/stockExample",
                                          "getStockPrice");
root.setNodeValue("DOW");
DOMSource request = new DOMSource(requestDoc);

// Dispatch disp created previously
DOMSource response = disp.invoke(request);
```

Asynchronous invocation

`Dispatch` objects also support asynchronous invocations. As with the higher level asynchronous APIs discussed in Chapter 40, *Developing Asynchronous Applications*, `Dispatch` objects can use both the polling approach and the callback approach.

When using the polling approach, the `invokeAsync()` method returns a `Response<t>` object that can be polled to see if the response has arrived. Example 41.5, "The `Dispatch.invokeAsync()` Method for Polling" shows the signature of the method used to make an asynchronous invocation using the polling approach.

Example 41.5. The `Dispatch.invokeAsync()` Method for Polling

```
Response <T> invokeAsync(T msg)
    throws WebServiceException;
```

For detailed information on using the polling approach for asynchronous invocations see Section 40.3, "Implementing an Asynchronous Client with the Polling Approach".

When using the callback approach, the `invokeAsync()` method takes an `AsyncHandler` implementation that processes the response when it is returned. Example 41.6, "The `Dispatch.invokeAsync()` Method Using a Callback" shows the signature of the method used to make an asynchronous invocation using the callback approach.

Example 41.6. The `Dispatch.invokeAsync()` Method Using a Callback

```
Future<?> invokeAsync(T msg,
                      AsyncHandler<T> handler)
    throws WebServiceException;
```

For detailed information on using the callback approach for asynchronous invocations see Section 40.4, "Implementing an Asynchronous Client with the Callback Approach" .

NOTE

As with the synchronous **invoke()** method, the type of the response and the type of the request are determined when you create the **Dispatch** object.

Oneway invocation

When a request does not generate a response, make remote invocations using the **Dispatch** object's **invokeOneWay()**. Example 41.7, "The **Dispatch.invokeOneWay()** Method" shows the signature for this method.

Example 41.7. The Dispatch.invokeOneWay() Method

```
void invokeOneWay(T msg)
    throws WebServiceException;
```

The type of object used to package the request is determined when the **Dispatch** object is created. For example if the **Dispatch** object is created using **createDispatch(portName, DOMSource.class, Service.Mode.PAYLOAD)**, then the request is packaged into a **DOMSource** object.

NOTE

When using JAXB objects, the response and the request can be of any type the provided **JAXBContext** object can marshal and unmarshal.

Example 41.8, "Making a One Way Invocation Using a **Dispatch** Object" shows code for making a oneway invocation on a remote service using a JAXB object.

Example 41.8. Making a One Way Invocation Using a Dispatch Object

```
// Creating a JAXBContext and an Unmarshaller for the request
JAXBContext jbc =
JAXBContext.newInstance("org.apache.cxf.StockExample");
Unmarshaller u = jbc.createUnmarshaller();

// Read the request from disk
File rf = new File("request.xml");
GetStockPrice request = (GetStockPrice)u.unmarshal(rf);

// Dispatch disp created previously
disp.invokeOneWay(request);
```

41.2. USING XML IN A SERVICE PROVIDER

The **Provider** interface is a low-level JAX-WS API that allows you to implement a service provider that works directly with messages as raw XML. The messages are not packaged into JAXB objects before being passed to an object that implements the **Provider** interface.

41.2.1. Messaging Modes

Overview

Objects that implement the **`Provider`** interface have two *messaging modes*:

- Message mode
- Payload mode

The messaging mode you specify determines the level of messaging detail that is passed to your implementation.

Message mode

When using *message mode*, a **`Provider`** implementation works with complete messages. A complete message includes any binding specific headers and wrappers. For example, a **`Provider`** implementation that uses a SOAP binding receives requests as fully specified SOAP message. Any response returned from the implementation must be a fully specified SOAP message.

To specify that a **`Provider`** implementation uses message mode by provide the value **`java.xml.ws.Service.Mode.MESSAGE`** as the value to the **`javax.xml.ws.ServiceMode`** annotation, as shown in Example 41.9, "Specifying that a **`Provider`** Implementation Uses Message Mode".

Example 41.9. Specifying that a `Provider` Implementation Uses Message Mode

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
public class stockQuoteProvider implements Provider<SOAPMessage>
{
  ...
}
```

Payload mode

In *payload mode* a **`Provider`** implementation works with only the payload of a message. For example, a **`Provider`** implementation working in payload mode works only with the body of a SOAP message. The binding layer processes any binding level wrappers and headers.

TIP

When working with a binding that does not use special wrappers, such as the Apache CXF XML binding, payload mode and message mode provide the same results.

To specify that a **`Provider`** implementation uses payload mode by provide the value **`java.xml.ws.Service.Mode.PAYLOAD`** as the value to the **`javax.xml.ws.ServiceMode`** annotation, as shown in Example 41.10, "Specifying that a **`Provider`** Implementation Uses Payload Mode".

Example 41.10. Specifying that a `Provider` Implementation Uses Payload Mode

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
```

```
public class stockQuoteProvider implements Provider<DOMSource>
{
  ...
}
```

TIP

If you do not provide a value for the **@ServiceMode** annotation, the **Provider** implementation uses payload mode.

41.2.2. Data Types

Overview

Because they are low-level objects, **Provider** implementations cannot use the same JAXB generated types as the higher level consumer APIs. **Provider** implementations work with the following types of objects:

- **javax.xml.transform.Source**
- **javax.xml.soap.SOAPMessage**
- **javax.activation.DataSource**

Using Source objects

A **Provider** implementation can accept and return objects that are derived from the **javax.xml.transform.Source** interface. **Source** objects are low level objects that hold XML documents. Each **Source** implementation provides methods that access the stored XML documents and manipulate its contents. The following objects implement the **Source** interface:

DOMSource

Holds XML messages as a Document Object Model(DOM) tree. The XML message is stored as a set of **Node** objects that are accessed using the **getNode()** method. Nodes can be either updated or added to the DOM tree using the **setNode()** method.

SAXSource

Holds XML messages as a Simple API for XML (SAX) object. SAX objects contain an **InputSource** object that holds the raw data and an **XMLReader** object that parses the raw data.

StreamSource

Holds XML messages as a data stream. The data stream can be manipulated the same as any other data stream.

If you create your **Provider** object so that it uses generic **Source** objects, Apache CXF returns the messages as **SAXSource** objects.

This behavior can be changed using the endpoint's source-preferred-format property. See Part IV, “Configuring Web Service Endpoints” for information about configuring the Apache CXF runtime.

IMPORTANT

When using **Source** objects the developer is responsible for ensuring that all required binding specific wrappers are added to the message. For example, when interacting with a service expecting SOAP messages, the developer must ensure that the required SOAP envelope is added to the outgoing request and that the SOAP envelope's contents are correct.

Using SOAPMessage objects

Provider implementations can use **javax.xml.soap.SOAPMessage** objects when the following conditions are true:

- The **Provider** implementation is using the SOAP binding
- The **Provider** implementation is using message mode

A **SOAPMessage** object holds a SOAP message. They contain one **SOAPPart** object and zero or more **AttachmentPart** objects. The **SOAPPart** object contains the SOAP specific portions of the SOAP message including the SOAP envelope, any SOAP headers, and the SOAP message body. The **AttachmentPart** objects contain binary data that is passed as an attachment.

Using DataSource objects

Provider implementations can use objects that implement the **javax.activation.DataSource** interface when the following conditions are true:

- The implementation is using the HTTP binding
- The implementation is using message mode

DataSource objects provide a mechanism for working with MIME typed data from a variety of sources, including URLs, files, and byte arrays.

41.2.3. Implementing a Provider Object

Overview

The **Provider** interface is relatively easy to implement. It only has one method, **invoke()**, that must be implemented. In addition it has three simple requirements:

- An implementation must have the **@WebServiceProvider** annotation.
- An implementation must have a default public constructor.
- An implementation must implement a typed version of the **Provider** interface.

  In other words, you cannot implement a **Provider<T>** interface. You must implement a version of the interface that uses a concrete data type as listed in Section 41.2.2, "Data Types". For example, you can implement an instance of a **Provider<SAXSource>**.

The complexity of implementing the **Provider** interface is in the logic handling the request messages and building the proper responses.

Working with messages

Unlike the higher-level SEI based service implementations, **Provider** implementations receive requests as raw XML data, and must send responses as raw XML data. This requires that the developer has intimate knowledge of the messages used by the service being implemented. These details can typically be found in the WSDL document describing the service.

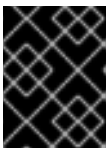
WS-I Basic Profile provides guidelines about the messages used by services, including:

- The root element of a request is based in the value of the **name** attribute of the **wsdl:operation** element that corresponds to the operation that is invoked.

WARNING

If the service uses doc/literal bare messages, the root element of the request is based on the value of **name** attribute of the **wsdl:part** element referred to by the **wsdl:operation** element.

- The root element of all messages is namespace qualified.
- If the service uses rpc/literal messages, the top-level elements in the messages are not namespace qualified.



IMPORTANT

The children of top-level elements might be namespace qualified, but to be certain you will must check their schema definitions.

- If the service uses rpc/literal messages, none of the top-level elements can be null.
- If the service uses doc/literal messages, then the schema definition of the message determines if any of the elements are namespace qualified.

The @WebServiceProvider annotation

To be recognized by JAX-WS as a service implementation, a **Provider** implementation must be decorated with the **@WebServiceProvider** annotation.

Table 41.2, "**@WebServiceProvider** Properties" describes the properties that can be set for the **@WebServiceProvider** annotation.

Table 41.2. @WebServiceProvider Properties

Property	Description
portName	Specifies the value of the **name** attribute of the **wsdl:port** element that defines the service's endpoint.

Property	Description
serviceName	Specifies the value of the **name** attribute of the **wsdl:service** element that contains the service's endpoint.
targetNamespace	Specifies the targetname space of the service's WSDL definition.
wsdlLocation	Specifies the URI for the WSDL document defining the service.

All of these properties are optional, and are empty by default. If you leave them empty, Apache CXF creates values using information from the implementation class.

Implementing the `invoke()` method

The **Provider** interface has only one method, **invoke()**, that must be implemented. The **invoke()** method receives the incoming request packaged into the type of object declared by the type of **Provider** interface being implemented, and returns the response message packaged into the same type of object. For example, an implementation of a **Provider<SOAPMessage>** interface receives the request as a **SOAPMessage** object and returns the response as a **SOAPMessage** object.

The messaging mode used by the **Provider** implementation determines the amount of binding specific information the request and the response messages contain. Implementations using message mode receive all of the binding specific wrappers and headers along with the request. They must also add all of the binding specific wrappers and headers to the response message. Implementations using payload mode only receive the body of the request. The XML document returned by an implementation using payload mode is placed into the body of the request message.

Examples

Example 41.11, "**Provider<SOAPMessage>** Implementation" shows a **Provider** implementation that works with **SOAPMessage** objects in message mode.

Example 41.11. Provider<SOAPMessage> Implementation

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

1 @WebServiceProvider(portName="stockQuoteReporterPort"
                      serviceName="stockQuoteReporter")
2 @ServiceMode(value="Service.Mode.MESSAGE")
  public class  stockQuoteReporterProvider implements
Provider<SOAPMessage>
{
3 public stockQuoteReporterProvider()
    {
  }
```

```
4 public SOAPMessage invoke(SOAPMessage request)
    {
5   SOAPBody requestBody = request.getSOAPBody();
6
    if(requestBody.getElementName.getLocalName.equals("getStockPrice"))
    {
7     MessageFactory mf = MessageFactory.newInstance();
        SOAPFactory sf = SOAPFactory.newInstance();

8     SOAPMessage response = mf.createMessage();
        SOAPBody respBody = response.getSOAPBody();
      Name bodyName = sf.createName("getStockPriceResponse");
      respBody.addBodyElement(bodyName);
      SOAPElement respContent = respBody.addChildElement("price");
      respContent.setValue("123.00");
      response.saveChanges();
9     return response;
      }
    ...
  }
}
```

The code in Example 41.11, "**Provider<SOAPMessage>** Implementation" does the following:

1. Specifies that the following class implements a **Provider** object that implements the service whose **wsdl:service** element is named **stockQuoteReporter**, and whose **wsdl:port** element is named **stockQuoteReporterPort**.
2. Specifies that this **Provider** implementation uses message mode.
3. Provides the required default public constructor.
4. Provides an implementation of the **invoke()** method that takes a **SOAPMessage** object and returns a **SOAPMessage** object.
5. Extracts the request message from the body of the incoming SOAP message.
6. Checks the root element of the request message to determine how to process the request.
7. Creates the factories required for building the response.
8. Builds the SOAP message for the response.
9. Returns the response as a **SOAPMessage** object.

Example 41.12, "**Provider<DOMSource>** Implementation" shows an example of a **Provider** implementation using **DOMSource** objects in payload mode.

Example 41.12. Provider<DOMSource> Implementation

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
```

```
import javax.xml.ws.WebServiceProvider;

1 @WebServiceProvider(portName="stockQuoteReporterPort"
  serviceName="stockQuoteReporter")
2 @ServiceMode(value="Service.Mode.PAYLOAD")
  public class  stockQuoteReporterProvider implements
Provider<DOMSource>
3 public stockQuoteReporterProvider()
    {
  }

4 public DOMSource invoke(DOMSource request)
    {
    DOMSource response = new DOMSource();
    ...
    return response;
  }
}
```

The code in Example 41.12, "**Provider<DOMSource>** Implementation" does the following:

1. Specifies that the class implements a **Provider** object that implements the service whose **wsdl:service** element is named **stockQuoteReporter**, and whose **wsdl:port** element is named **stockQuoteReporterPort**.
2. Specifies that this **Provider** implementation uses payload mode.
3. Provides the required default public constructor.
4. Provides an implementation of the **invoke()** method that takes a **DOMSource** object and returns a **DOMSource** object.

CHAPTER 42. WORKING WITH CONTEXTS

Abstract

JAX-WS uses contexts to pass metadata along the messaging chain. This metadata, depending on its scope, is accessible to implementation level code. It is also accessible to JAX-WS handlers that operate on the message below the implementation level.

42.1. UNDERSTANDING CONTEXTS

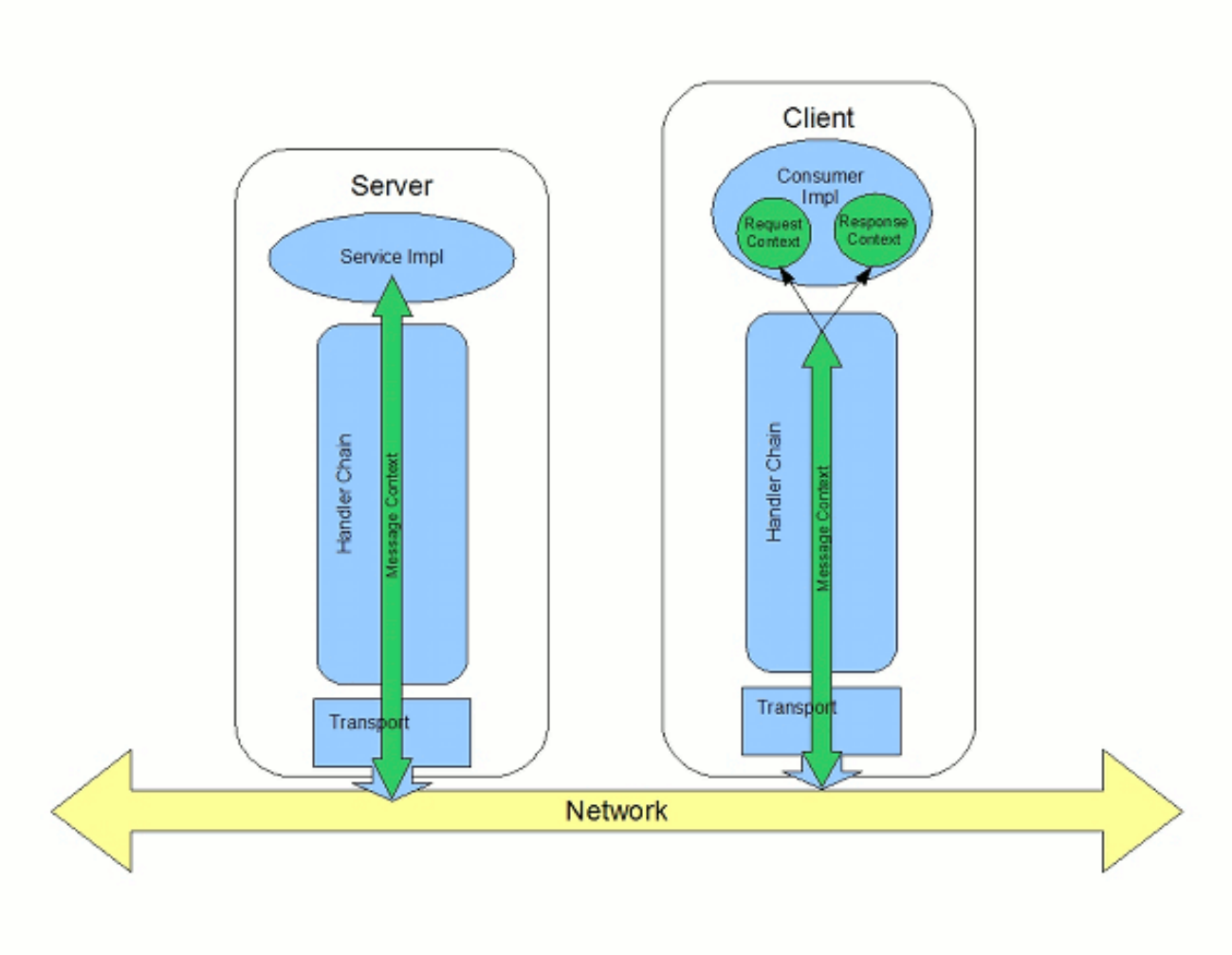
Overview

In many instances it is necessary to pass information about a message to other parts of an application. Apache CXF does this using a context mechanism. Contexts are maps that hold properties relating to an outgoing or an incoming message. The properties stored in the context are typically metadata about the message, and the underlying transport used to communicate the message. For example, the transport specific headers used in transmitting the message, such as the HTTP response code or the JMS correlation ID, are stored in the JAX-WS contexts.

The contexts are available at all levels of a JAX-WS application. However, they differ in subtle ways depending upon where in the message processing stack you are accessing the context. JAX-WS **`Handler`** implementations have direct access to the contexts and can access all properties that are set in them. Service implementations access contexts by having them injected, and can only access properties that are set in the **`APPLICATION`** scope. Consumer implementations can only access properties that are set in the **`APPLICATION`** scope.

Figure 42.1, “Message Contexts and Message Processing Path” shows how the context properties pass through Apache CXF. As a message passes through the messaging chain, its associated message context passes along with it.

Figure 42.1. Message Contexts and Message Processing Path



How properties are stored in a context

The message contexts are all implementations of the `javax.xml.ws.handler.MessageContext` interface. The `MessageContext` interface extends the `java.util.Map<String key, Object value>` interface. `Map` objects store information as key value pairs.

In a message context, properties are stored as name/value pairs. A property's key is a `String` that identifies the property. The value of a property can be any value stored in any Java object. When the value is returned from a message context, the application must know the type to expect and cast accordingly. For example, if a property's value is stored in a `UserInfo` object it is still returned from a message context as an `Object` object that must be cast back into a `UserInfo` object.

Properties in a message context also have a scope. The scope determines where a property can be accessed in the message processing chain.

Property scopes

Properties in a message context are scoped. A property can be in one of the following scopes:

`APPLICATION`

Properties scoped as `APPLICATION` are available to JAX-WS `Handler` implementations, consumer implementation code, and service provider implementation code. If a handler needs to pass a property to the service provider implementation, it sets the property's scope to `APPLICATION`. All

properties set from either the consumer implementation or the service provider implementation contexts are automatically scoped as **APPLICATION**.

HANDLER

Properties scoped as **HANDLER** are only available to JAX-WS **Handler** implementations. Properties stored in a message context from a **Handler** implementation are scoped as **HANDLER** by default.

You can change a property's scope using the message context's **setScope()** method. Example 42.1, "The **MessageContext.setScope()** Method" shows the method's signature.

Example 42.1. The MessageContext.setScope() Method

```
void setScope(String key,
              MessageContext.Scope scope)
    throws java.lang.IllegalArgumentException;
```

The first parameter specifies the property's key. The second parameter specifies the new scope for the property. The scope can be either:

- **MessageContext.Scope.APPLICATION**
- **MessageContext.Scope.HANDLER**

Overview of contexts in handlers

Classes that implement the JAX-WS **Handler** interface have direct access to a message's context information. The message's context information is passed into the **Handler** implementation's **handleMessage()**, **handleFault()**, and **close()** methods.

Handler implementations have access to all of the properties stored in the message context, regardless of their scope. In addition, logical handlers use a specialized message context called a **LogicalMessageContext**. **LogicalMessageContext** objects have methods that access the contents of the message body.

Overview of contexts in service implementations

Service implementations can access properties scoped as **APPLICATION** from the message context. The service provider's implementation object accesses the message context through the **WebServiceContext** object.

For more information see Section 42.2, "Working with Contexts in a Service Implementation" .

Overview of contexts in consumer implementations

Consumer implementations have indirect access to the contents of the message context. The consumer implementation has two separate message contexts:

- Request context — holds a copy of the properties used for outgoing requests
- Response context — holds a copy of the properties from an incoming response

The dispatch layer transfers the properties between the consumer implementation's message contexts

and the message context used by the **Handler** implementations.

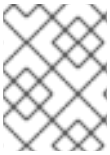
When a request is passed to the dispatch layer from the consumer implementation, the contents of the request context are copied into the message context that is used by the dispatch layer. When the response is returned from the service, the dispatch layer processes the message and sets the appropriate properties into its message context. After the dispatch layer processes a response, it copies all of the properties scoped as **APPLICATION** in its message context to the consumer implementation's response context.

For more information see Section 42.3, “Working with Contexts in a Consumer Implementation” .

42.2. WORKING WITH CONTEXTS IN A SERVICE IMPLEMENTATION

Overview

Context information is made available to service implementations using the **WebServiceContext** interface. From the **WebServiceContext** object you can obtain a **MessageContext** object that is populated with the current request's context properties in the application scope. You can manipulate the values of the properties, and they are propagated back through the response chain.



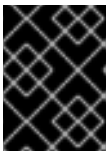
NOTE

The **MessageContext** interface inherits from the **java.util.Map** interface. Its contents can be manipulated using the **Map** interface's methods.

Obtaining a context

To obtain the message context in a service implementation do the following:

1. Declare a variable of type **WebServiceContext**.
2. Decorate the variable with the **javax.annotation.Resource** annotation to indicate that the context information is being injected into the variable.
3. Obtain the **MessageContext** object from the **WebServiceContext** object using the **getMessageContext()** method.



IMPORTANT

getMessageContext() can only be used in methods that are decorated with the **@WebMethod** annotation.

Example 42.2, “Obtaining a Context Object in a Service Implementation” shows code for obtaining a context object.

Example 42.2. Obtaining a Context Object in a Service Implementation

```
import javax.xml.ws.*;
import javax.xml.ws.handler.*;
import javax.annotation.*;

@WebServiceProvider
public class WidgetServiceImpl
```

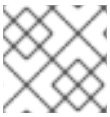
```
{
  @Resource
  WebServiceContext wsc;

  @WebMethod
  public String getColor(String itemNum)
  {
    MessageContext context = wsc.getMessageContext();
  }

  ...
}
```

Reading a property from a context

Once you have obtained the **MessageContext** object for your implementation, you can access the properties stored there using the **get()** method shown in Example 42.3, "The `MessageContext.get()` Method".

Example 42.3. The `MessageContext.get()` Method

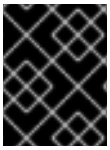
```
V get(Object key);
```



NOTE

This **get()** is inherited from the **Map** interface.

The ***key*** parameter is the string representing the property you want to retrieve from the context. The **get()** returns an object that must be cast to the proper type for the property. Table 42.1, "Properties Available in the Service Implementation Context" lists a number of the properties that are available in a service implementation's context.



IMPORTANT

Changing the values of the object returned from the context also changes the value of the property in the context.

Example 42.4, "Getting a Property from a Service's Message Context" shows code for getting the name of the WSDL **operation** element that represents the invoked operation.

Example 42.4. Getting a Property from a Service's Message Context

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

  ...
  // MessageContext context retrieved in a previous example
  QName wsdl_operation = (QName)context.get(Message.WSDL_OPERATION);
```

Setting properties in a context

Once you have obtained the **MessageContext** object for your implementation, you can set properties, and change existing properties, using the **put()** method shown in Example 42.5, "The MessageContext.put() Method".

Example 42.5. The MessageContext.put() Method

```
V put(K key,
      V value)
    throws ClassCastException, IllegalArgumentException,
NullPointerException;
```

If the property being set already exists in the message context, the **put()** method replaces the existing value with the new value and returns the old value. If the property does not already exist in the message context, the **put()** method sets the property and returns **null**.

Example 42.6, "Setting a Property in a Service's Message Context" shows code for setting the response code for an HTTP request.

Example 42.6. Setting a Property in a Service's Message Context

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

  ...
  // MessageContext context retrieved in a previous example
  context.put(Message.RESPONSE_CODE, new Integer(404));
```

Supported contexts

Table 42.1, "Properties Available in the Service Implementation Context" lists the properties accessible through the context in a service implementation object.

Table 42.1. Properties Available in the Service Implementation Context

Base Class Property Name	Description
org.apache.cxf.message.Message	
PROTOCOL_HEADERS[a]	Specifies the transport specific header information. The value is stored as a **java.util.Map<String, List<String>>**.
RESPONSE_CODE[a]	Specifies the response code returned to the consumer. The value is stored as an **Integer** object.

Base Class Property Name	Description
ENDPOINT_ADDRESS	Specifies the address of the service provider. The value is stored as a **String**.
HTTP_REQUEST_METHOD[a]	Specifies the HTTP verb sent with a request. The value is stored as a **String**.
PATH_INFO[a]	Specifies the path of the resource being requested. The value is stored as a **String**. The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URI is http://cxf.apache.org/demo/widgets the path is **/demo/widgets**.
QUERY_STRING[a]	Specifies the query, if any, attached to the URI used to invoke the request. The value is stored as a **String**. Queries appear at the end of the URI after a **?**. For example, if a request is made to http://cxf.apache.org/demo/widgets?color the query is **color**.
MTOM_ENABLED	Specifies whether or not the service provider can use MTOM for SOAP attachments. The value is stored as a **Boolean**.
SCHEMA_VALIDATION_ENABLED	Specifies whether or not the service provider validates messages against a schema. The value is stored as a **Boolean**.
FAULT_STACKTRACE_ENABLED	Specifies if the runtime provides a stack trace along with a fault message. The value is stored as a **Boolean**.
CONTENT_TYPE	Specifies the MIME type of the message. The value is stored as a **String**.
BASE_PATH	Specifies the path of the resource being requested. The value is stored as a **java.net.URL**. The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URL is http://cxf.apache.org/demo/widgets the base path is **/demo/widgets**.
ENCODING	Specifies the encoding of the message. The value is stored as a **String**.

Base Class Property Name	Description
FIXED_PARAMETER_ORDER	Specifies whether the parameters must appear in the message in a particular order. The value is stored as a **Boolean**.
MAINTAIN_SESSION	Specifies if the consumer wants to maintain the current session for future requests. The value is stored as a **Boolean**.
WSDL_DESCRIPTION[a]	Specifies the WSDL document that defines the service being implemented. The value is stored as a **org.xml.sax.InputSource** object.
WSDL_SERVICE[a]	Specifies the qualified name of the **wsdl:service** element that defines the service being implemented. The value is stored as a **QName**.
WSDL_PORT[a]	Specifies the qualified name of the **wsdl:port** element that defines the endpoint used to access the service. The value is stored as a **QName**.
WSDL_INTERFACE[a]	Specifies the qualified name of the **wsdl:portType** element that defines the service being implemented. The value is stored as a **QName**.
WSDL_OPERATION[a]	Specifies the qualified name of the **wsdl:operation** element that corresponds to the operation invoked by the consumer. The value is stored as a **QName**.
javax.xml.ws.handler.MessageContext	
MESSAGE_OUTBOUND_PROPERTY	Specifies if a message is outbound. The value is stored as a **Boolean**. **true** specifies that a message is outbound.
INBOUND_MESSAGE_ATTACHMENTS	Contains any attachments included in the request message. The value is stored as a **java.util.Map<String, DataHandler>**. The key value for the map is the MIME Content-ID for the header.
OUTBOUND_MESSAGE_ATTACHMENTS	Contains any attachments for the response message. The value is stored as a **java.util.Map<String, DataHandler>**. The key value for the map is the MIME Content-ID for the header.

Base Class	
Property Name	**Description**
WSDL_DESCRIPTION	Specifies the WSDL document that defines the service being implemented. The value is stored as a **org.xml.sax.InputSource** object.
WSDL_SERVICE	Specifies the qualified name of the **wsdl:service** element that defines the service being implemented. The value is stored as a **QName**.
WSDL_PORT	Specifies the qualified name of the **wsdl:port** element that defines the endpoint used to access the service. The value is stored as a **QName**.
WSDL_INTERFACE	Specifies the qualified name of the **wsdl:portType** element that defines the service being implemented. The value is stored as a **QName**.
WSDL_OPERATION	Specifies the qualified name of the **wsdl:operation** element that corresponds to the operation invoked by the consumer. The value is stored as a **QName**.
HTTP_RESPONSE_CODE	Specifies the response code returned to the consumer. The value is stored as an **Integer** object.
HTTP_REQUEST_HEADERS	Specifies the HTTP headers on a request. The value is stored as a **java.util.Map<String, List<String>>**.
HTTP_RESPONSE_HEADERS	Specifies the HTTP headers for the response. The value is stored as a **java.util.Map<String, List<String>>**.
HTTP_REQUEST_METHOD	Specifies the HTTP verb sent with a request. The value is stored as a **String**.
SERVLET_REQUEST	Contains the servlet's request object. The value is stored as a **javax.servlet.http.HttpServletRequest**.
SERVLET_RESPONSE	Contains the servlet's response object. The value is stored as a **javax.servlet.http.HttpResponse**.

Base Class Property Name	Description
SERVLET_CONTEXT	Contains the servlet's context object. The value is stored as a **javax.servlet.ServletContext**.
PATH_INFO	Specifies the path of the resource being requested. The value is stored as a **String**. The path is the portion of the URI after the hostname and before any query string. For example, if an endpoint's URL is http://cxf.apache.org/demo/widgets the path is **/demo/widgets**.
QUERY_STRING	Specifies the query, if any, attached to the URI used to invoke the request. The value is stored as a **String**. Queries appear at the end of the URI after a **?**. For example, if a request is made to http://cxf.apache.org/demo/widgets?color the query string is **color**.
REFERENCE_PARAMETERS	Specifies the WS-Addressing reference parameters. This includes all of the SOAP headers whose **wsa:IsReferenceParameter** attribute is set to **true**. The value is stored as a **java.util.List**.
org.apache.cxf.transport.jms.JMSConstants	
JMS_SERVER_HEADERS	Contains the JMS message headers. For more information see Section 42.4, "Working with JMS Message Properties".
[a] When using HTTP this property is the same as the standard JAX-WS defined property.	

42.3. WORKING WITH CONTEXTS IN A CONSUMER IMPLEMENTATION

Overview

Consumer implementations have access to context information through the **BindingProvider** interface. The **BindingProvider** instance holds context information in two separate contexts:

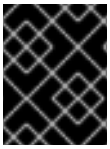
Request Context

The *request context* enables you to set properties that affect outbound messages. Request context properties are applied to a specific port instance and, once set, the properties affect every subsequent operation invocation made on the port, until such time as a property is explicitly

cleared. For example, you might use a request context property to set a connection timeout or to initialize data for sending in a header.

Response Context

The *response context* enables you to read the property values set by the response to the last operation invocation made from the current thread. Response context properties are reset after every operation invocation. For example, you might access a response context property to read header information received from the last inbound message.



IMPORTANT

Only information that is placed in the application scope of a message context can be accessed by the consumer implementation.

Obtaining a context

Contexts are obtained using the `javax.xml.ws.BindingProvider` interface. The `BindingProvider` interface has two methods for obtaining a context:

`getRequestContext()`

The `getRequestContext()` method, shown in Example 42.7, "The `getRequestContext()` Method", returns the request context as a `Map` object. The returned `Map` object can be used to directly manipulate the contents of the context.

Example 42.7. The `getRequestContext()` Method

```
Map<String, Object> getRequestContext();
```

`getResponseContext()`

The `getResponseContext()`, shown in Example 42.8, "The `getResponseContext()` Method", returns the response context as a `Map` object. The returned `Map` object's contents reflect the state of the response context's contents from the most recent successful request on a remote service made in the current thread.

Example 42.8. The `getResponseContext()` Method

```
Map<String, Object> getResponseContext();
```

Since proxy objects implement the `BindingProvider` interface, a `BindingProvider` object can be obtained by casting a proxy object. The contexts obtained from the `BindingProvider` object are only valid for operations invoked on the proxy object used to create it.

Example 42.9, "Getting a Consumer's Request Context" shows code for obtaining the request context for a proxy.

Example 42.9. Getting a Consumer's Request Context

```
// Proxy widgetProxy obtained previously
BindingProvider bp = (BindingProvider)widgetProxy;
Map<String, Object> responseContext = bp.getResponseContext();
```

Reading a property from a context

Consumer contexts are stored in **`java.util.Map<String, Object>`** objects. The map has keys that are String objects and values that contain arbitrary objects. Use **`java.util.Map.get()`** to access an entry in the map of response context properties.

To retrieve a particular context property, *ContextPropertyName*, use the code shown in Example 42.10, "Reading a Response Context Property".

Example 42.10. Reading a Response Context Property

```
// Invoke an operation.
port.SomeOperation();

// Read response context property.
java.util.Map<String, Object> responseContext =
  ((javax.xml.ws.BindingProvider)port).getResponseContext();
PropertyType propValue = (PropertyType)
responseContext.get(ContextPropertyName);
```

Setting properties in a context

Consumer contexts are hash maps stored in **`java.util.Map<String, Object>`** objects. The map has keys that are String objects and values that are arbitrary objects. To set a property in a context use the **`java.util.Map.put()`** method.

TIP

While you can set properties in both the request context and the response context, only the changes made to the request context have any impact on message processing. The properties in the response context are reset when each remote invocation is completed on the current thread.

The code shown in Example 42.11, "Setting a Request Context Property" changes the address of the target service provider by setting the value of the **`BindingProvider.ENDPOINT_ADDRESS_PROPERTY`**.

Example 42.11. Setting a Request Context Property

```
// Set request context property.
java.util.Map<String, Object> requestContext =
     ((javax.xml.ws.BindingProvider)port).getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
"http://localhost:8080/widgets");

// Invoke an operation.
port.SomeOperation();
```

IMPORTANT

Once a property is set in the request context its value is used for all subsequent remote invocations. You can change the value and the changed value will then be used.

Supported contexts

Apache CXF supports the following context properties in consumer implementations:

Table 42.2. Consumer Context Properties

Base Class Property Name	Description
`javax.xml.ws.BindingProvider`	
`ENDPOINT_ADDRESS_PROPERTY`	Specifies the address of the target service. The value is stored as a **`String`**.
`USERNAME_PROPERTY`[a]	Specifies the username used for HTTP basic authentication. The value is stored as a **`String`**.
`PASSWORD_PROPERTY`[b]	Specifies the password used for HTTP basic authentication. The value is stored as a **`String`**.
`SESSION_MAINTAIN_PROPERTY`[c]	Specifies if the client wants to maintain session information. The value is stored as a **`Boolean`** object.
org.apache.cxf.ws.addressing.JAXWSAConstants	
`CLIENT_ADDRESSING_PROPERTIES`	Specifies the WS-Addressing information used by the consumer to contact the desired service provider. The value is stored as a **`org.apache.cxf.ws.addressing.AddressingProperties`**.
org.apache.cxf.transports.jms.context.JMSConstants	
`JMS_CLIENT_REQUEST_HEADERS`	Contains the JMS headers for the message. For more information see Section 42.4, “Working with JMS Message Properties”.

[a] This property is overridden by the username defined in the HTTP security settings.

[b] This property is overridden by the password defined in the HTTP security settings.

[c] The Apache CXF ignores this property.

Base Class	
Property Name	**Description**

42.4. WORKING WITH JMS MESSAGE PROPERTIES

The Apache CXF JMS transport has a context mechanism that can be used to inspect a JMS message's properties. The context mechanism can also be used to set a JMS message's properties.

42.4.1. Inspecting JMS Message Headers

Consumers and services use different context mechanisms to access the JMS message header properties. However, both mechanisms return the header properties as a **`org.apache.cxf.transports.jms.context.JMSMessageHeadersType`** object.

Getting the JMS Message Headers in a Service

To get the JMS message header properties from the **`WebServiceContext`** object, do the following:

1. Obtain the context as described in the section called "Obtaining a context".
2. Get the message headers from the message context using the message context's **`get()`** method with the parameter **`org.apache.cxf.transports.jms.JMSConstants.JMS_SERVER_HEADERS`**.

Example 42.12, "Getting JMS Message Headers in a Service Implementation" shows code for getting the JMS message headers from a service's message context:

Example 42.12. Getting JMS Message Headers in a Service Implementation

```
import org.apache.cxf.transport.jms.JMSConstants;
import org.apache.cxf.transports.jms.context.JMSMessageHeadersType;

@WebService(serviceName = "HelloWorldService",
                          portName = "HelloWorldPort",
                          endpointInterface =
"org.apache.cxf.hello_world_jms.HelloWorldPortType",
                          targetNamespace =
"http://cxf.apache.org/hello_world_jms")
  public class GreeterImplTwoWayJMS implements HelloWorldPortType
  {
    @Resource
    protected WebServiceContext wsContext;
    ...

    @WebMethod
```

```
    public String greetMe(String me)
    {
      MessageContext mc = wsContext.getMessageContext();
      JMSMessageHeadersType headers = (JMSMessageHeadersType)
mc.get(JMSConstants.JMS_SERVER_HEADERS);
       ...
     }
      ...
}
```

Getting JMS Message Header Properties in a Consumer

Once a message is successfully retrieved from the JMS transport you can inspect the JMS header properties using the consumer's response context. In addition, you can see how long the client waits for a response before timing out.

You can To get the JMS message headers from a consumer's response context do the following:

1. Get the response context as described in the section called “Obtaining a context”.
2. Get the JMS message header properties from the response context using the context's **`get()`** method with **`org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RESPONSE_HEADERS`** as the parameter.

Example 42.13, “Getting the JMS Headers from a Consumer Response Header” shows code for getting the JMS message header properties from a consumer's response context.

Example 42.13. Getting the JMS Headers from a Consumer Response Header

```
import org.apache.cxf.transports.jms.context.*;
// Proxy greeter initialized previously
1 BindingProvider  bp = (BindingProvider)greeter;
  2 Map<String, Object> responseContext = bp.getResponseContext();
    3 JMSMessageHeadersType responseHdr = (JMSMessageHeadersType)

responseContext.get(JMSConstants.JMS_CLIENT_REQUEST_HEADERS);
...
}
```

The code in Example 42.13, “Getting the JMS Headers from a Consumer Response Header” does the following:

1. Casts the proxy to a **`BindingProvider`**.
2. Gets the response context.
3. Retrieves the JMS message headers from the response context.

42.4.2. Inspecting the Message Header Properties

Standard JMS Header Properties

Table 42.3, "JMS Header Properties" lists the standard properties in the JMS header that you can inspect.

Table 42.3. JMS Header Properties

Property Name	Property Type	Getter Method
Correlation ID	string	`getJMSCorralationID()`
Delivery Mode	int	`getJMSDeliveryMode()`
Message Expiration	long	`getJMSExpiration()`
Message ID	string	`getJMSMessageID()`
Priority	int	`getJMSPriority()`
Redelivered	boolean	`getJMSRedlivered()`
Time Stamp	long	`getJMSTimeStamp()`
Type	string	`getJMSType()`
Time To Live	long	`getTimeToLive()`

Optional Header Properties

In addition, you can inspect any optional properties stored in the JMS header using `JMSMessageHeadersType.getProperty()`. The optional properties are returned as a `List` of `org.apache.cxf.transports.jms.context.JMSPropertyType`. Optional properties are stored as name/value pairs.

Example

Example 42.14, "Reading the JMS Header Properties" shows code for inspecting some of the JMS properties using the response context.

Example 42.14. Reading the JMS Header Properties

```
// JMSMessageHeadersType messageHdr retrieved previously
1 System.out.println("Correlation ID:
   "+messageHdr.getJMSCorrelationID());
2 System.out.println("Message Priority: "+messageHdr.getJMSPriority());
  3 System.out.println("Redelivered: "+messageHdr.getRedelivered());

JMSPropertyType prop = null;
4 List<JMSPropertyType> optProps = messageHdr.getProperty();
  5 Iterator<JMSPropertyType> iter = optProps.iterator();
    6 while (iter.hasNext())
```

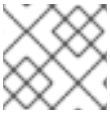
```
{
  prop = iter.next();
   System.out.println("Property name: "+prop.getName());
   System.out.println("Property value: "+prop.getValue());
}
```

The code in Example 42.14, “Reading the JMS Header Properties” does the following:

1. Prints the value of the message's correlation ID.
2. Prints the value of the message's priority property.
3. Prints the value of the message's redelivered property.
4. Gets the list of the message's optional header properties.
5. Gets an `Iterator` to traverse the list of properties.
6. Iterates through the list of optional properties and prints their name and value.

42.4.3. Setting JMS Properties

Using the request context in a consumer endpoint, you can set a number of the JMS message header properties and the consumer endpoint's timeout value. These properties are valid for a single invocation. You must reset them each time you invoke an operation on the service proxy.



NOTE

You cannot set header properties in a service.

JMS Header Properties

Table 42.4, “Settable JMS Header Properties” lists the properties in the JMS header that can be set using the consumer endpoint's request context.

Table 42.4. Settable JMS Header Properties

Property Name	Property Type	Setter Method
Correlation ID	string	`setJMSCorralationID()`
Delivery Mode	int	`setJMSDeliveryMode()`
Priority	int	`setJMSPriority()`
Time To Live	long	`setTimeToLive()`

To set these properties do the following:

1. Create an `org.apache.cxf.transports.jms.context.JMSMessageHeadersType` object.

2. Populate the values you want to set using the appropriate setter methods described in Table 42.4, "Settable JMS Header Properties".

3. Set the values to the request context by calling the request context's **put()** method using **org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_REQUEST_HEADERS** as the first argument, and the new **JMSMessageHeadersType** object as the second argument.

Optional JMS Header Properties

You can also set optional properties to the JMS header. Optional JMS header properties are stored in the **JMSMessageHeadersType** object that is used to set the other JMS header properties. They are stored as a **List** object containing **org.apache.cxf.transports.jms.context.JMSPropertyType** objects. To add optional properties to the JMS header do the following:

1. Create a **JMSPropertyType** object.

2. Set the property's name field using **setName()**.

3. Set the property's value field using **setValue()**.

4. Add the property to the JMS message header using **JMSMessageHeadersType.getProperty().add(JMSPropertyType)**.

5. Repeat the procedure until all of the properties have been added to the message header.

Client Receive Timeout

In addition to the JMS header properties, you can set the amount of time a consumer endpoint waits for a response before timing out. You set the value by calling the request context's **put()** method with **org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RECEIVE_TIMEOUT** as the first argument and a long representing the amount of time in milliseconds that you want the consumer to wait as the second argument.

Example

Example 42.15, "Setting JMS Properties using the Request Context" shows code for setting some of the JMS properties using the request context.

Example 42.15. Setting JMS Properties using the Request Context

```
import org.apache.cxf.transports.jms.context.*;
 // Proxy greeter initialized previously
1 InvocationHandler handler = Proxy.getInvocationHandler(greeter);


BindingProvider bp= null;
2 if (handler instanceof BindingProvider)
  {
3   bp = (BindingProvider)handler;
4     Map<String, Object> requestContext = bp.getRequestContext();

5   JMSMessageHeadersType requestHdr = new JMSMessageHeadersType();
6     requestHdr.setJMSCorrelationID("WithBob");
        requestHdr.setJMSExpiration(3600000L);
```

```
7
8   JMSPropertyType prop = new JMSPropertyType;
9     prop.setName("MyProperty");
      prop.setValue("Bluebird");
10  requestHdr.getProperty().add(prop);

11  requestContext.put(JMSConstants.CLIENT_REQUEST_HEADERS,
    requestHdr);

12  requestContext.put(JMSConstants.CLIENT_RECEIVE_TIMEOUT, new
    Long(1000));
}
```

The code in Example 42.15, "Setting JMS Properties using the Request Context" does the following:

1. Gets the **`InvocationHandler`** for the proxy whose JMS properties you want to change.
2. Checks to see if the **`InvocationHandler`** is a **`BindingProvider`**.
3. Casts the returned **`InvocationHandler`** object into a **`BindingProvider`** object to retrieve the request context.
4. Gets the request context.
5. Creates a **`JMSMessageHeadersType`** object to hold the new message header values.
6. Sets the Correlation ID.
7. Sets the Expiration property to 60 minutes.
8. Creates a new **`JMSPropertyType`** object.
9. Sets the values for the optional property.
10. Adds the optional property to the message header.
11. Sets the JMS message header values into the request context.
12. Sets the client receive timeout property to 1 second.

CHAPTER 43. WRITING HANDLERS

Abstract

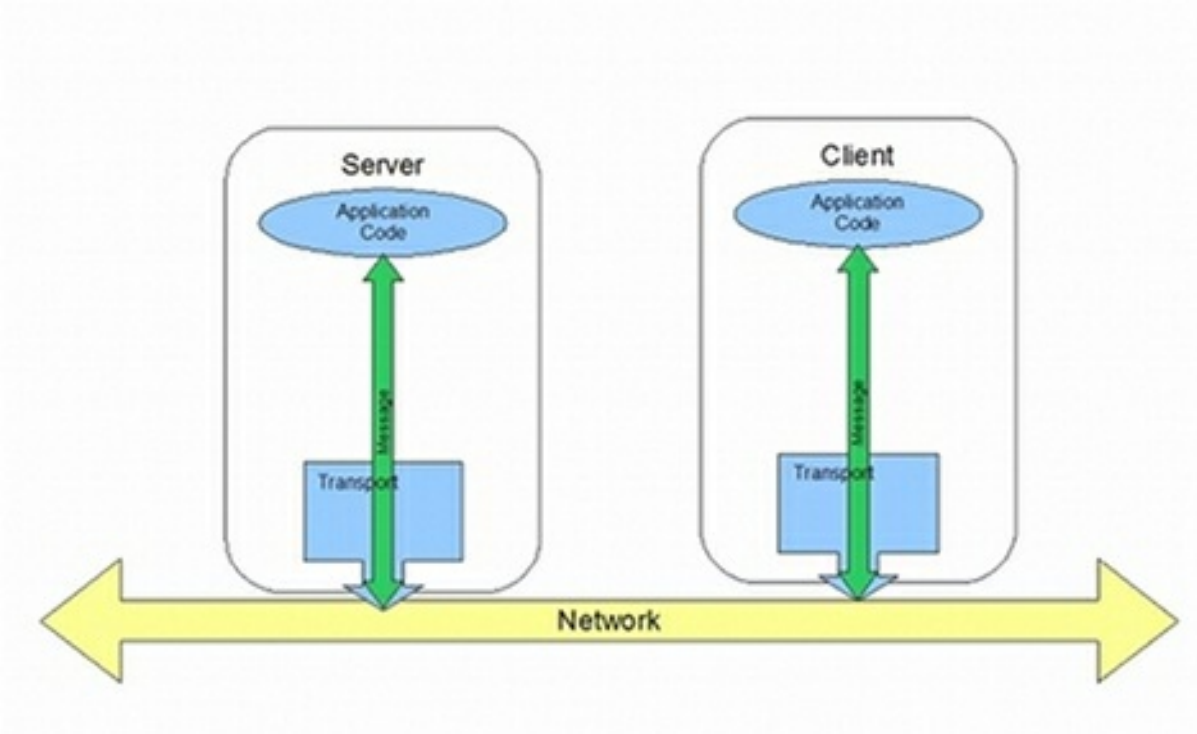
JAX-WS provides a flexible plug-in framework for adding message processing modules to an application. These modules, known as handlers, are independent of the application level code and can provide low-level message processing capabilities.

43.1. HANDLERS: AN INTRODUCTION

Overview

When a service proxy invokes an operation on a service, the operation's parameters are passed to Apache CXF where they are built into a message and placed on the wire. When the message is received by the service, Apache CXF reads the message from the wire, reconstructs the message, and then passes the operation parameters to the application code responsible for implementing the operation. When the application code is finished processing the request, the reply message undergoes a similar chain of events on its trip to the service proxy that originated the request. This is shown in Figure 43.1, “Message Exchange Path”.

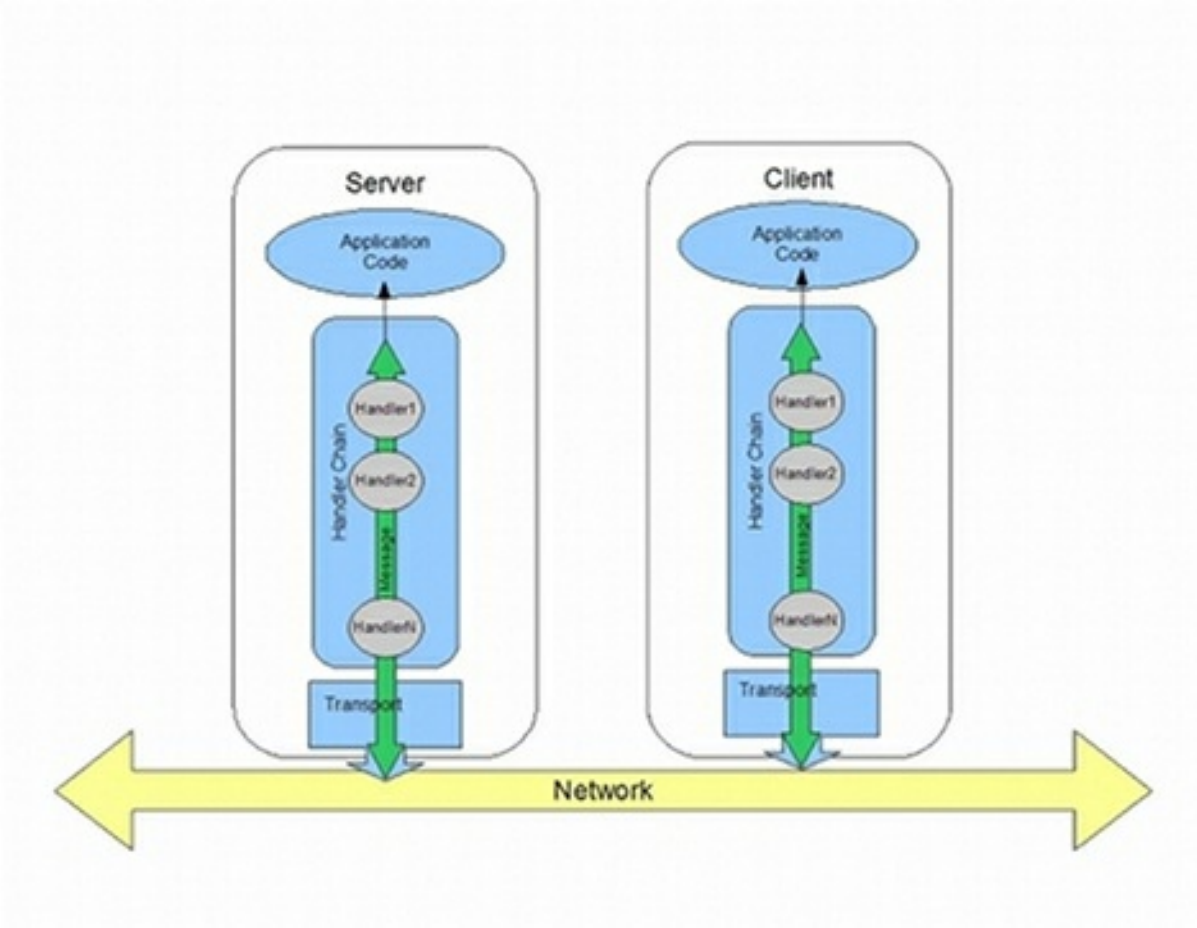
Figure 43.1. Message Exchange Path



JAX-WS defines a mechanism for manipulating the message data between the application level code and the network. For example, you might want the message data passed over the open network to be encrypted using a proprietary encryption mechanism. You could write a JAX-WS handler that encrypted and decrypted the data. Then you could insert the handler into the message processing chains of all clients and servers.

As shown in Figure 43.2, “Message Exchange Path with Handlers” , the handlers are placed in a chain that is traversed between the application level code and the transport code that places the message onto the network.

Figure 43.2. Message Exchange Path with Handlers



Handler types

The JAX-WS specification defines two basic handler types:

Logical Handler

Logical handlers can process the message payload and the properties stored in the message context. For example, if the application uses pure XML messages, the logical handlers have access to the entire message. If the application uses SOAP messages, the logical handlers have access to the contents of the SOAP body. They do not have access to either the SOAP headers or any attachments unless they were placed into the message context.

Logical handlers are placed closest to the application code on the handler chain. This means that they are executed first when a message is passed from the application code to the transport. When a message is received from the network and passed back to the application code, the logical handlers are executed last.

Protocol Handler

Protocol handlers can process the entire message received from the network and the properties stored in the message context. For example, if the application uses SOAP messages, the protocol handlers would have access to the contents of the SOAP body, the SOAP headers, and any attachments.

Protocol handlers are placed closest to the transport on the handler chain. This means that they are executed first when a message is received from the network. When a message is sent to the network from the application code, the protocol handlers are executed last.

TIP

The only protocol handler supported by Apache CXF is specific to SOAP.

Implementation of handlers

The differences between the two handler types are very subtle and they share a common base interface. Because of their common parentage, logical handlers and protocol handlers share a number of methods that must be implemented, including:

handleMessage()

The `handleMessage()` method is the central method in any handler. It is the method responsible for processing normal messages.

handleFault()

`handleFault()` is the method responsible for processing fault messages.

close()

`close()` is called on all executed handlers in a handler chain when a message has reached the end of the chain. It is used to clean up any resources consumed during message processing.

The differences between the implementation of a logical handler and the implementation of a protocol handler revolve around the following:

- The specific interface that is implemented

  All handlers implement an interface that derives from the `Handler` interface. Logical handlers implement the `LogicalHandler` interface. Protocol handlers implement protocol specific extensions of the `Handler` interface. For example, SOAP handlers implement the `SOAPHandler` interface.

- The amount of information available to the handler

  Protocol handlers have access to the contents of messages and all of the protocol specific information that is packaged with the message content. Logical handlers can only access the contents of the message. Logical handlers have no knowledge of protocol details.

Adding handlers to an application

To add a handler to an application you must do the following:

1. Determine whether the handler is going to be used on the service providers, the consumers, or both.
2. Determine which type of handler is the most appropriate for the job.
3. Implement the proper interface.

   To implement a logical handler see Section 43.2, “Implementing a Logical Handler”.

   To implement a protocol handler see Section 43.4, “Implementing a Protocol Handler”.

4. Configure your endpoint(s) to use the handlers.

43.2. IMPLEMENTING A LOGICAL HANDLER

Overview

Logical handlers implement the `javax.xml.ws.handler.LogicalHandler` interface. The `LogicalHandler` interface, shown in Example 43.1, "`LogicalHandler` Synopsis" passes a `LogicalMessageContext` object to the `handleMessage()` method and the `handleFault()` method. The context object provides access to the *body* of the message and to any properties set into the message exchange's context.

Example 43.1. `LogicalHandler` Synopsis

```
public interface LogicalHandler extends Handler
{
  boolean handleMessage(LogicalMessageContext context);
  boolean handleFault(LogicalMessageContext context);
  void close(LogicalMessageContext context);
}
```

Procedure

To implement a logical hander you do the following:

1. Implement any initialization logic required by the handler.
2. Implement the message handling logic.
3. Implement the fault handling logic.
4. Implement the logic for closing the handler when it is finished.
5. Implement any logic for cleaning up the handler's resources before it is destroyed.

43.3. HANDLING MESSAGES IN A LOGICAL HANDLER

Overview

Normal message processing is handled by the `handleMessage()` method.

The `handleMessage()` method receives a `LogicalMessageContext` object that provides access to the message body and any properties stored in the message context.

The `handleMessage()` method returns either `true` or `false` depending on how message processing is to continue. It can also throw an exception.

Getting the message data

The `LogicalMessageContext` object passed into logical message handlers allows access to the message body using the context's `getMessage()` method. The `getMessage()` method, shown in

Example 43.2, "Method for Getting the Message Payload in a Logical Handler" , returns the message payload as a **LogicalMessage** object.

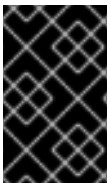
Example 43.2. Method for Getting the Message Payload in a Logical Handler

```
LogicalMessage getMessage();
```

Once you have the **LogicalMessage** object, you can use it to manipulate the message body. The **LogicalMessage** interface, shown in Example 43.3, "Logical Message Holder", has getters and setters for working with the actual message body.

Example 43.3. Logical Message Holder

```
LogicalMessage {
 Source getPayload();
 Object getPayload(JAXBContext context);
 void setPayload(Object payload,
                 JAXBContext context);
 void setPayload(Source payload);
}
```



IMPORTANT

The contents of the message payload are determined by the type of binding in use. The SOAP binding only allows access to the SOAP body of the message. The XML binding allows access to the entire message body.

Working with the message body as an XML object

One pair of getters and setters of the logical message work with the message payload as a **javax.xml.transform.dom.DOMSource** object.

The **getPayload()** method that has no parameters returns the message payload as a **DOMSource** object. The returned object is the actual message payload. Any changes made to the returned object change the message body immediately.

You can replace the body of the message with a **DOMSource** object using the **setPayload()** method that takes the single **Source** object.

Working with the message body as a JAXB object

The other pair of getters and setters allow you to work with the message payload as a JAXB object. They use a **JAXBContext** object to transform the message payload into JAXB objects.

To use the JAXB objects you do the following:

1. Get a **JAXBContext** object that can manage the data types in the message body.

   For information on creating a **JAXBContext** object see Chapter 38, *Using A **JAXBContext** Object*.

2. Get the message body as shown in Example 43.4.

Example 43.4. Getting the Message Body as a JAXB Object

```
JAXBContext jaxbc = JAXBContext(myObjectFactory.class);
Object body = message.getPayload(jaxbc);
```

3. Cast the returned object to the proper type.
4. Manipulate the message body as needed.
5. Put the updated message body back into the context as shown in Example 43.5.

Example 43.5. Updating the Message Body Using a JAXB Object

```
message.setPayload(body, jaxbc);
```

Working with context properties

The logical message context passed into a logical handler is an instance of the application's message context and can access all of the properties stored in it. Handlers have access to properties at both the **APPLICATION** scope and the **HANDLER** scope.

Like the application's message context, the logical message context is a subclass of Java **Map**. To access the properties stored in the context, you use the **get()** method and **put()** method inherited from the **Map** interface.

By default, any properties you set in the message context from inside a logical handler are assigned a scope of **HANDLER**. If you want the application code to be able to access the property you need to use the context's **setScope()** method to explicitly set the property's scope to **APPLICATION**.

For more information on working with properties in the message context see Section 42.1, “Understanding Contexts”.

Determining the direction of the message

It is often important to know the direction a message is passing through the handler chain. For example, you would want to retrieve a security token from incoming requests and attach a security token to an outgoing response.

The direction of the message is stored in the message context's outbound message property. You retrieve the outbound message property from the message context using the **MessageContext.MESSAGE_OUTBOUND_PROPERTY** key as shown in Example 43.6, “Getting the Message's Direction from the SOAP Message Context”.

Example 43.6. Getting the Message's Direction from the SOAP Message Context

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

The property is stored as a **Boolean** object. You can use the object's **booleanValue()** method to determine the property's value. If the property is set to **true**, the message is outbound. If the property is set to **false** the message is inbound.

Determining the return value

How the **handleMessage()** method completes its message processing has a direct impact on how message processing proceeds. It can complete by doing one of the following actions:

I. Return **true**—Returning **true** signals to the Apache CXF runtime that message processing should continue normally. The next handler, if any, has its **handleMessage()** invoked.

II. Return **false**—Returning **false** signals to the Apache CXF runtime that normal message processing must stop. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

   For request-response message exchanges the following happens:

   1. The direction of message processing is reversed.

      For example, if a request is being processed by a service provider, the message stops progressing toward the service's implementation object. Instead, it is sent back towards the binding for return to the consumer that originated the request.

   2. Any message handlers that reside along the handler chain in the new processing direction have their **handleMessage()** method invoked in the order in which they reside in the chain.

   3. When the message reaches the end of the handler chain it is dispatched.

   For one-way message exchanges the following happens:

   1. Message processing stops.

   2. All previously invoked message handlers have their **close()** method invoked.

   3. The message is dispatched.

III. Throw a **ProtocolException** exception—Throwing a **ProtocolException** exception, or a subclass of this exception, signals the Apache CXF runtime that fault message processing is beginning. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

   For request-response message exchanges the following happens:

   1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.

   2. The direction of message processing is reversed.

      For example, if a request is being processed by a service provider, the message stops progressing toward the service's implementation object. Instead, it is sent back towards the binding for return to the consumer that originated the request.

   3. Any message handlers that reside along the handler chain in the new processing direction have their **handleFault()** method invoked in the order in which they reside in the chain.

4. When the fault message reaches the end of the handler chain it is dispatched.

For one-way message exchanges the following happens:

1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.
2. Message processing stops.
3. All previously invoked message handlers have their `close()` method invoked.
4. The fault message is dispatched.

IV. Throw any other runtime exception—Throwing a runtime exception other than a `ProtocolException` exception signals the Apache CXF runtime that message processing is to stop. All previously invoked message handlers have the `close()` method invoked and the exception is dispatched. If the message is part of a request-response message exchange, the exception is dispatched so that it is returned to the consumer that originated the request.

Example

Example 43.7, "Logical Message Handler Message Processing" shows an implementation of `handleMessage()` message for a logical message handler that is used by a service consumer. It processes requests before they are sent to the service provider.

Example 43.7. Logical Message Handler Message Processing

```
public class SmallNumberHandler implements
LogicalHandler<LogicalMessageContext>
{
    public final boolean handleMessage(LogicalMessageContext
messageContext)
    {
        try
        {
            boolean outbound =
(Boolean)messageContext.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

1           if (outbound)
            {
2               LogicalMessage msg = messageContext.getMessage();

              JAXBContext jaxbContext =
JAXBContext.newInstance(ObjectFactory.class);
3               Object payload = msg.getPayload(jaxbContext);
                if (payload instanceof JAXBElement)
              {
                  payload = ((JAXBElement)payload).getValue();
              }

4               if (payload instanceof AddNumbers)
                {
                  AddNumbers req = (AddNumbers)payload;

                  int a = req.getArg0();
                  int b = req.getArg1();
```

```
                    int answer = a + b;

5                     if (answer < 20)
                      {
                      AddNumbersResponse resp = new
6 AddNumbersResponse();
                        resp.setReturn(answer);
                      msg.setPayload(new
ObjectFactory().createAddNumbersResponse(resp),
                                                        jaxbContext);

7                     return false;
                      }
                }
                else
                {
8                     throw new WebServiceException("Bad Request");
                  }
              }
9               return true;
            }
10          catch (JAXBException ex)
            {
              throw new ProtocolException(ex);
          }
      }
...
}
```

The code in Example 43.7, “Logical Message Handler Message Processing” does the following:

1. Checks if the message is an outbound request.

   If the message is an outbound request, the handler does additional message processing.

2. Gets the **`LogicalMessage`** representation of the message payload from the message context.

3. Gets the actual message payload as a JAXB object.

4. Checks to make sure the request is of the correct type.

   If it is, the handler continues processing the message.

5. Checks the value of the sum.

   If it is less than the threshold of 20 then it builds a response and returns it to the client.

6. Builds the response.

7. Returns **`false`** to stop message processing and return the response to the client.

8. Throws a runtime exception if the message is not of the correct type.

   This exception is returned to the client.

9 Returns **true** if the message is an inbound response or the sum does not meet the threshold.

Message processing continues normally.

10 Throws a **ProtocolException** if a JAXB marshalling error is encountered.

The exception is passed back to the client after it is processed by the **handleFault()** method of the handlers between the current handler and the client.

43.4. IMPLEMENTING A PROTOCOL HANDLER

Overview

Protocol handlers are specific to the protocol in use. Apache CXF provides the SOAP protocol handler as specified by JAX-WS. A SOAP protocol handler implements the **javax.xml.ws.handler.soap.SOAPHandler** interface.

The **SOAPHandler** interface, shown in Example 43.8, "**SOAPHandler** Synopsis", uses a SOAP specific message context that provides access to the message as a **SOAPMessage** object. It also allows you to access the SOAP headers.

Example 43.8. SOAPHandler Synopsis

```
public interface SOAPHandler extends Handler
{
  boolean handleMessage(SOAPMessageContext context);
  boolean handleFault(SOAPMessageContext context);
  void close(SOAPMessageContext context);
  Set<QName> getHeaders()
}
```

In addition to using a SOAP specific message context, SOAP protocol handlers require that you implement an additional method called **getHeaders()**. This additional method returns the QNames of the header blocks the handler can process.

Procedure

To implement a logical hander do the following:

1. Implement any initialization logic required by the handler.
2. Implement the message handling logic.
3. Implement the fault handling logic.
4. Implement the **getHeaders()** method.
5. Implement the logic for closing the handler when it is finished.
6. Implement any logic for cleaning up the handler's resources before it is destroyed.

Implementing the `getHeaders()` method

The `getHeaders()`, shown in Example 43.9, "The `SOAPHander.getHeaders()` Method", method informs the Apache CXF runtime what SOAP headers the handler is responsible for processing. It returns the QNames of the outer element of each SOAP header the handler understands.

Example 43.9. The `SOAPHander.getHeaders()` Method

```
Set<QName> getHeaders();
```

For many cases simply returning `null` is sufficient. However, if the application uses the `mustUnderstand` attribute of any of the SOAP headers, then it is important to specify the headers understood by the application's SOAP handlers. The runtime checks the set of SOAP headers that all of the registered handlers understand against the list of headers with the `mustUnderstand` attribute set to `true`. If any of the flagged headers are not in the list of understood headers, the runtime rejects the message and throws a SOAP must understand exception.

43.5. HANDLING MESSAGES IN A SOAP HANDLER

Overview

Normal message processing is handled by the `handleMessage()` method.

The `handleMessage()` method receives a `SOAPMessageHandler` object that provides access to the message body as a `SOAPMessage` object and the SOAP headers associated with the message. In addition, the context provides access to any properties stored in the message context.

The `handleMessage()` method returns either `true` or `false` depending on how message processing is to continue. It can also throw an exception.

Working with the message body

You can get the SOAP message using the SOAP message context's `getMessage()` method. It returns the message as a live `SOAPMessage` object. Any changes to the message in the handler are automatically reflected in the message stored in the context.

If you wish to replace the existing message with a new one, you can use the context's `setMessage()` method. The `setMessage()` method takes a `SOAPMessage` object.

Getting the SOAP headers

You can access the SOAP message's headers using the `SOAPMessage` object's `getHeader()` method. This will return the SOAP header as a `SOAPHeader` object that you will need to inspect to find the header elements you wish to process.

The SOAP message context provides a `getHeaders()` method, shown in Example 43.10, "The `SOAPMessageContext.getHeaders()` Method", that will return an array containing JAXB objects for the specified SOAP headers.

Example 43.10. The `SOAPMessageContext.getHeaders()` Method

```
Ojbect[] getHeaders(QName header,
                    JAXBContext context,
                    boolean allRoles);
```

You specify the headers using the QName of their element. You can further limit the headers that are returned by setting the ***allRoles*** parameter to **false**. That instructs the runtime to only return the SOAP headers that are applicable to the active SOAP roles.

If no headers are found, the method returns an empty array.

For more information about instantiating a **JAXBContext** object see Chapter 38, *Using A **JAXBContext** Object*.

Working with context properties

The SOAP message context passed into a logical handler is an instance of the application's message context and can access all of the properties stored in it. Handlers have access to properties at both the **APPLICATION** scope and the **Handler** scope.

Like the application's message context, the SOAP message context is a subclass of Java **Map**. To access the properties stored in the context, you use the **get()** method and **put()** method inherited from the **Map** interface.

By default, any properties you set in the context from inside a logical handler will be assigned a scope of **HANDLER**. If you want the application code to be able to access the property you need to use the context's **setScope()** method to explicitly set the property's scope to **APPLICATION**.

For more information on working with properties in the message context see Section 42.1, "Understanding Contexts".

Determining the direction of the message

It is often important to know the direction a message is passing through the handler chain. For example, you would want to add headers to an outgoing message and strip headers from an incoming message.

The direction of the message is stored in the message context's outbound message property. You retrieve the outbound message property from the message context using the **MessageContext.MESSAGE_OUTBOUND_PROPERTY** key as shown in Example 43.11, "Getting the Message's Direction from the SOAP Message Context".

Example 43.11. Getting the Message's Direction from the SOAP Message Context

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

The property is stored as a **Boolean** object. You can use the object's **booleanValue()** method to determine the property's value. If the property is set to **true**, the message is outbound. If the property is set to **false** the message is inbound.

Determining the return value

How the `handleMessage()` method completes its message processing has a direct impact on how message processing proceeds. It can complete by doing one of the following actions:

I. return `true`—Returning `true` signals to the Apache CXF runtime that message processing should continue normally. The next handler, if any, has its `handleMessage()` invoked.

II. return `false`—Returning `false` signals to the Apache CXF runtime that normal message processing is to stop. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

    For request-response message exchanges the following happens:

    1. The direction of message processing is reversed.

       For example, if a request is being processed by a service provider, the message will stop progressing toward the service's implementation object. It will instead be sent back towards the binding for return to the consumer that originated the request.

    2. Any message handlers that reside along the handler chain in the new processing direction have their `handleMessage()` method invoked in the order in which they reside in the chain.

    3. When the message reaches the end of the handler chain it is dispatched.

    For one-way message exchanges the following happens:

    1. Message processing stops.

    2. All previously invoked message handlers have their `close()` method invoked.

    3. The message is dispatched.

III. throw a `ProtocolException` exception—Throwing a `ProtocolException` exception, or a subclass of this exception, signals the Apache CXF runtime that fault message processing is to start. How the runtime proceeds depends on the message exchange pattern in use for the *current message*.

    For request-response message exchanges the following happens:

    1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.

    2. The direction of message processing is reversed.

       For example, if a request is being processed by a service provider, the message will stop progressing toward the service's implementation object. It will be sent back towards the binding for return to the consumer that originated the request.

    3. Any message handlers that reside along the handler chain in the new processing direction have their `handleFault()` method invoked in the order in which they reside in the chain.

    4. When the fault message reaches the end of the handler chain it is dispatched.

    For one-way message exchanges the following happens:

    1. If the handler has not already created a fault message, the runtime wraps the message in a fault message.

2. Message processing stops.
3. All previously invoked message handlers have their `close()` method invoked.
4. The fault message is dispatched.

IV. throw any other runtime exception—Throwing a runtime exception other than a `ProtocolException` exception signals the Apache CXF runtime that message processing is to stop. All previously invoked message handlers have the `close()` method invoked and the exception is dispatched. If the message is part of a request-response message exchange the exception is dispatched so that it is returned to the consumer that originated the request.

Example

Example 43.12, "Handling a Message in a SOAP Handler" shows a `handleMessage()` implementation that prints the SOAP message to the screen.

Example 43.12. Handling a Message in a SOAP Handler

```
public boolean handleMessage(SOAPMessageContext smc)
{
  PrintStream out;

  Boolean outbound =
(1)(Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

(2)  if (outbound.booleanValue())
     {
    out.println("\nOutbound message:");
  }
  else
  {
    out.println("\nInbound message:");
  }

(3)  SOAPMessage message = smc.getMessage();

(4)  message.writeTo(out);
     out.println();

  return true;
}
```

The code in Example 43.12 does the following:

1. Retrieves the outbound property from the message context.
2. Tests the messages direction and prints the appropriate message.
3. Retrieves the SOAP message from the context.
4. Prints the message to the console.

43.6. INITIALIZING A HANDLER

Overview

When the runtime creates an instance of a handler, it creates all of the resources the hander needs to process messages. While you can place all of the logic for doing this in the handler's constructor, it may not be the most appropriate place. The handler framework performs a number of optional steps when it instantiates a handler. You can add resource injection and other initialization logic that will be executed during the optional steps.

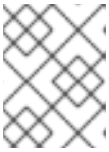
TIP

You do not have to provide any initialization methods for a handler.

Order of initialization

The Apache CXF runtime initializes a handler in the following manner:

1. The handler's constructor is called.
2. Any resources that are specified by the **`@Resource`** annotation are injected.
3. The method decorated with **`@PostConstruct`** annotation, if it is present, is called.



NOTE

Methods decorated with the **`@PostConstruct`** annotation must have a `void` return type and have no parameters.

4. The handler is place in the **Ready** state.

43.7. HANDLING FAULT MESSAGES

Overview

Handlers use the **`handleFault()`** method for processing fault messages when a **`ProtocolException`** exception is thrown during message processing.

The **`handleFault()`** method receives either a **`LogicalMessageContext`** object or **`SOAPMessageContext`** object depending on the type of handler. The received context gives the handler's implementation access to the message payload.

The **`handleFault()`** method returns either **`true`** or **`false`**, depending on how fault message processing is to proceed. It can also throw an exception.

Getting the message payload

The context object received by the **`handleFault()`** method is similar to the one received by the **`handleMessage()`** method. You use the context's **`getMessage()`** method to access the message payload in the same way. The only difference is the payload contained in the context.

For more information on working with a **LogicalMessageContext** see Section 43.3, "Handling Messages in a Logical Handler".

For more information on working with a **SOAPMessageContext** see Section 43.5, "Handling Messages in a SOAP Handler".

Determining the return value

How the **handleFault()** method completes its message processing has a direct impact on how message processing proceeds. It completes by performing one of the following actions:

Return true

Returning **true** signals that fault processing should continue normally. The **handleFault()** method of the next handler in the chain will be invoked.

Return false

Returning **false** signals that fault processing stops. The **close()** method of the handlers that were invoked in processing the current message are invoked and the fault message is dispatched.

Throw an exception

Throwing an exception stops fault message processing. The **close()** method of the handlers that were invoked in processing the current message are invoked and the exception is dispatched.

Example

Example 43.13, "Handling a Fault in a Message Handler" shows an implementation of **handleFault()** that prints the message body to the screen.

Example 43.13. Handling a Fault in a Message Handler

```
public final boolean handleFault(LogicalMessageContext messageContext)
{
  System.out.println("handleFault() called with message:");

  LogicalMessage msg=messageContext.getMessage();
  System.out.println(msg.getPayload());

  return true;
}
```

43.8. CLOSING A HANDLER

When a handler chain is finished processing a message, the runtime calls each executed handler's **close()** method. This is the appropriate place to clean up any resources that were used by the handler during message processing or resetting any properties to a default state.

If a resource needs to persist beyond a single message exchange, you should not clean it up during in the handler's **close()** method.

43.9. RELEASING A HANDLER

Overview

The runtime releases a handler when the service or service proxy to which the handler is bound is shutdown. The runtime will invoke an optional release method before invoking the handler's destructor. This optional release method can be used to release any resources used by the handler or perform other actions that would not be appropriate in the handler's destructor.

TIP

You do not have to provide any clean-up methods for a handler.

Order of release

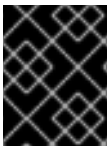
The following happens when the handler is released:

1. The handler finishes processing any active messages.
2. The runtime invokes the method decorated with the `@PreDestroy` annotation.

   This method should clean up any resources used by the handler.
3. The handler's destructor is called.

43.10. CONFIGURING ENDPOINTS TO USE HANDLERS

43.10.1. Programmatic Configuration



IMPORTANT

Any handler chains configured using the Spring configuration override the handler chains configured programmaticaly.

43.10.1.1. Adding a Handler Chain to a Consumer

Overview

Adding a handler chain to a consumer involves explicitly building the chain of handlers. Then you set the handler chain directly on the service proxy's `Binding` object.

Procedure

To add a handler chain to a consumer you do the following:

1. Create a `List<Handler>` object to hold the handler chain.
2. Create an instance of each handler that will be added to the chain.
3. Add each of the instantiated handler objects to the list in the order they are to be invoked by the runtime.
4. Get the `Binding` object from the service proxy.

TIP

Apache CXF provides an implementation of the **Binding** interface called **org.apache.cxf.jaxws.binding.DefaultBindingImpl**.

5. Set the handler chain on the proxy using the **Binding** object's **setHandlerChain()** method.

Example

Example 43.14, "Adding a Handler Chain to a Consumer" shows code for adding a handler chain to a consumer.

Example 43.14. Adding a Handler Chain to a Consumer

```
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.Handler;
import java.util.ArrayList;
import java.util.List;

import org.apache.cxf.jaxws.binding.DefaultBindingImpl;
...
1 SmallNumberHandler sh = new SmallNumberHandler();
2 List<Handler> handlerChain = new ArrayList<Handler>();
3 handlerChain.add(sh);

4 DefaultBindingImpl binding = ((BindingProvider)proxy).getBinding();
5 binding.getBinding().setHandlerChain(handlerChain);
```

The code in Example 43.14, "Adding a Handler Chain to a Consumer" does the following:

1. Instantiates a handler.
2. Creates a **List** object to hold the chain.
3. Adds the handler to the chain.
4. Gets the **Binding** object from the proxy as a **DefaultBindingImpl** object.
5. Assigns the handler chain to the proxy's binding.

43.10.1.2. Adding a Handler Chain to a Service Provider

Overview

You add a handler chain to a service provider by decorating either the SEI or the implementation class with the **@HandlerChain** annotation. The annotation points to a meta-data file defining the handler chain used by the service provider.

Procedure

To add handler chain to a service provider you do the following:

1. Decorate the provider's implementation class with the **@HandlerChain** annotation.
2. Create a handler configuration file that defines the handler chain.

The @HandlerChain annotation

The **javax.jws.HandlerChain** annotation decorates service provider's implementation class. It instructs the runtime to load the handler chain configuration file specified by its file property.

The annotation's file property supports two methods for identifying the handler configuration file to load:

- a URL
- a relative path name

Example 43.15, "Service Implementation that Loads a Handler Chain" shows a service provider implementation that will use the handler chain defined in a file called **handlers.xml**. **handlers.xml** must be located in the directory from which the service provider is run.

Example 43.15. Service Implementation that Loads a Handler Chain

```
import javax.jws.HandlerChain;
import javax.jws.WebService;
...

@WebService(name = "AddNumbers",
            targetNamespace = "http://apache.org/handlers",
            portName = "AddNumbersPort",
            endpointInterface = "org.apache.handlers.AddNumbers",
            serviceName = "AddNumbersService")
@HandlerChain(file = "handlers.xml")
public class AddNumbersImpl implements AddNumbers
{
...
}
```

Handler configuration file

The handler configuration file defines a handler chain using the XML grammar that accompanies JSR 109(Web Services for Java EE, Version 1.2). This grammar is defined in the **http://java.sun.com/xml/ns/javaee**.

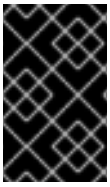
The root element of the handler configuration file is the **handler-chains** element. The **handler-chains** element has one or more **handler-chain** elements.

The **handler-chain** element define a handler chain. Table 43.1, "Elements Used to Define a Server-Side Handler Chain" describes the **handler-chain** element's children.

Table 43.1. Elements Used to Define a Server-Side Handler Chain

Element	Description
handler	Contains the elements that describe a handler.
service-name-pattern	Specifies the QName of the WSDL **service** element defining the service to which the handler chain is bound. You can use * as a wildcard when defining the QName.
port-name-pattern	Specifies the QName of the WSDL **port** element defining the endpoint to which the handler chain is bound. You can use * as a wildcard when defining the QName.
protocol-binding	Specifies the message binding for which the handler chain is used. The binding is specified as a URI or using one of the following aliases: **##SOAP11_HTTP**, **##SOAP11_HTTP_MTOM**, **##SOAP12_HTTP**, **##SOAP12_HTTP_MTOM**, or **##XML_HTTP**. For more information about message binding URIs see Appendix C, *Apache CXF Binding IDs*.

The **handler-chain** element is only required to have a single **handler** element as a child. It can, however, support as many **handler** elements as needed to define the complete handler chain. The handlers in the chain are executed in the order they specified in the handler chain definition.



IMPORTANT

The final order of execution will be determined by sorting the specified handlers into logical handlers and protocol handlers. Within the groupings, the order specified in the configuration will be used.

The other children, such as **protocol-binding**, are used to limit the scope of the defined handler chain. For example, if you use the **service-name-pattern** element, the handler chain will only be attached to service providers whose WSDL **port** element is a child of the specified WSDL **service** element. You can only use one of these limiting children in a **handler** element.

The **handler** element defines an individual handler in a handler chain. Its **handler-class** child element specifies the fully qualified name of the class implementing the handler. The **handler** element can also have an optional **handler-name** element that specifies a unique name for the handler.

Example 43.16, "Handler Configuration File" shows a handler configuration file that defines a single handler chain. The chain is made up of two handlers.

Example 43.16. Handler Configuration File

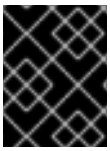
```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
```

```
      <handler-name>LoggingHandler</handler-name>
      <handler-class>demo.handlers.common.LoggingHandler</handler-
class>
    </handler>
    <handler>
      <handler-name>AddHeaderHandler</handler-name>
      <handler-class>demo.handlers.common.AddHeaderHandler</handler-
class>
    </handler>
  </handler-chain>
</handler-chains>
```

43.10.2. Spring Configuration

Overview

The easiest way to configure an endpoint to use a handler chain is to define the chain in the endpoint's configuration. This is done by adding a **jaxwxs:handlers** child to the element configuring the endpoint.



IMPORTANT

A handler chain added through the configuration file takes precedence over a handler chain configured programatically.

Procedure

To configure an endpoint to load a handler chain you do the following:

1. If the endpoint does not already have a configuration element, add one.

   For more information on configuring Apache CXF endpoints see Chapter 15, *Configuring JAX-WS Endpoints*.

2. Add a **jaxws:handlers** child element to the endpoint's configuration element.

3. For each handler in the chain, add a **bean** element specifying the class that implements the handler.

   TIP

   If your handler implementation is used in more than one place you can reference a **bean** element using the **ref** element.

The handlers element

The **jaxws:handlers** element defines a handler chain in an endpoint's configuration. It can appear as a child to all of the JAX-WS endpoint configuration elements. These are:

- **jaxws:endpoint** configures a service provider.
- **jaxws:server** also configures a service provider.

- **jaxws:client** configures a service consumer.

You add handlers to the handler chain in one of two ways:

- add a **bean** element defining the implementation class
- use a **ref** element to refer to a named **bean** element from elsewhere in the configuration file

The order in which the handlers are defined in the configuration is the order in which they will be executed. The order may be modified if you mix logical handlers and protocol handlers. The run time will sort them into the proper order while maintaining the basic order specified in the configuration.

Example

Example 43.17, “Configuring an Endpoint to Use a Handler Chain In Spring” shows the configuration for a service provider that loads a handler chain.

Example 43.17. Configuring an Endpoint to Use a Handler Chain In Spring

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="HandlerExample"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo">
    <jaxws:handlers>
       <bean class="demo.handlers.common.LoggingHandler" />
       <bean class="demo.handlers.common.AddHeaderHandler" />
    </jaxws:handlers>
  </jaws:endpoint>
</beans>
```

APPENDIX E. MAVEN TOOLING REFERENCE

NAME

Plug-in Setup — before you can use the Apache CXF plug-ins, you must first add the proper dependencies and repositories to your POM.

DEPENDENCIES

You need to add the following dependencies to your project's POM:

- the JAX-WS frontend

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-frontend-jaxws</artifactId>
  <version>version</version>
</dependency>
```

- the HTTP transport

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http</artifactId>
  <version>version</version>
</dependency>
```

- the Jetty transport

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-jetty</artifactId>
  <version>version</version>
</dependency>
```

REPOSITORIES

To ensure that you are using the Progress versions of the plug-ins you need to add the Apache CXF repositories to the project's POM:

- the plug-in repository

```
<pluginRepository>
  <id>fusesource.m2</id>
  <name>Apache CXF Open Source Community Release Repository</name>
  <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
  <releases>
    <enabled>true</enabled>
  </releases>
</pluginRepository>
```

- the Apache CXF release repository

```
<repository>
  <id>fusesource.m2</id>
  <name>Apache CXF Open Source Community Release Repository</name>
  <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
  <releases>
    <enabled>true</enabled>
  </releases>
</repository>
```

- the Apache CXF snapshot repository

```
<repository>
  <id>fusesource.m2-snapshot</id>
  <name>Apache CXF Open Source Community Snapshot Repository</name>
  <url>http://repo.fusesource.com/nexus/content/groups/public-
snapshots/</url>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
  <releases>
    <enabled>false</enabled>
  </releases>
</repository>
```

NAME

cxf-codegen-plugin — generates JAX-WS compliant Java code from a WSDL document

SYNOPSIS

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>version</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <configuration>
        <defaultOptions>
          <option>...</option>
          ...
        </defaultOptions>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>wsdlPath</wsdl>
            <option>...</option>
            ...
          </wsdlOption>
          ...
        </wsdlOptions>
```

```
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

DESCRIPTION

The **wsdl2java** task takes a WSDL document and generates fully annotated Java code from which to implement a service. The WSDL document must have a valid **portType** element, but it does not need to contain a **binding** element or a **service** element. Using the optional arguments you can customize the generated code.

WSDL OPTIONS

At least one **wsdlOptions** element is required to configure the plug-in. The **wsdlOptions** element's **wsdl** child is required and specifies a WSDL document to be processed by the plug-in. In addition to the **wsdl** element, the **wsdlOptions** element can take a number of children that can customize how the WSDL document is processed.

TIP

More than one **wsdlOptions** element can be listed in the plug-in configuration. Each element configures a single WSDL document for processing.

DEFAULT OPTIONS

The **defaultOptions** element is an optional element. It can be used to set options that are used across all of the specified WSDL documents.

IMPORTANT

If an option is duplicated in the **wsdlOptions** element, the value in the **wsdlOptions** element takes precedent.

OPTIONS

The options used to manage the code generation process are reviewed in the following table.

Option	Interpretation
-?	Displays the online help for this utility.
-help	
-h	
-fe *frontend*	Specifies the front end used by the code generator. The default is **jaxws**.[a]
-db *databinding*	Specifies the data binding used by the code generator. The default is **jaxb**.[b]

Option	Interpretation
-wv *wsdlVersion*	Specifies the WSDL version expected by the tool. The default is **1.1**.[c]
-p [*wsdlNamespace*=]*PackageName*	Specifies zero, or more, package names to use for the generated code. Optionally specifies the WSDL namespace to package name mapping.
-b *bindingName*	Specifies zero, or more, JAXWS or JAXB binding files. Use spaces to separate multiple entries.
-sn *serviceName*	Specifies the name of the WSDL service for which code is to be generated. The default is to generate code for every service in the WSDL document.
-d *output-directory*	Specifies the directory into which the generated code files are written.
-catalog *catalogUrl*	Specifies the URL of an XML catalog to use for resolving imported schemas and WSDL documents.
-compile	Compiles generated Java files.
-classdir *complile-class-dir*	Specifies the directory into which the compiled class files are written.
-client	Generates starting point code for a client mainline.
-server	Generates starting point code for a server mainline.
-impl	Generates starting point code for an implementation object.
-all	Generates all starting point code: types, service proxy, service interface, server mainline, client mainline, implementation object, and an Ant **build.xml** file.
-ant	Generates the Ant **build.xml** file.
-keep	Instructs the tool to not overwrite any existing files.
-defaultValues[=*DefaultValueProvider*]	Instructs the tool to generate default values for the generated client and the generated implementation. Optionally, you can also supply the name of the class used to generate the default values. By default, the **RandomValueProvider** class is used.

Option	Interpretation
-nexclude *schema-namespace*[=*java-packagename*]	Ignore the specified WSDL schema namespace when generating code. This option may be specified multiple times. Also, optionally specifies the Java package name used by types described in the excluded namespace(s).
-exsh (**true**/**false**)	Enables or disables processing of extended soap header message binding. Default is **false**.
-dns (**true**/**false**)	Enables or disables the loading of the default namespace package name mapping. Default is **true**.
-dex (**true**/**false**)	Enables or disables the loading of the default excludes namespace mapping. Default is **true**.
-wsdlLocation ***wsdlLocation***	Specifies the value of the**@WebService** annotation's wsdlLocation property.
-xjc*args*	Specifies a comma separated list of arguments to be passed to directly to the XJC when the JAXB data binding is being used. To get a list of all possible XJC arguments use the **-xjc-X**.
-noAddressBinding	Instructs the tool to use the Apache CXF proprietary WS-Addressing type instead of the JAX-WS 2.1 compliant mapping.
-validate	Instructs the tool to validate the WSDL document before attempting to generate any code.
-v	Displays the version number for the tool.
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.
wsdlfile	The path and name of the WSDL file to use in generating the code.

[a] Currently, Apache CXF only provides the JAX-WS front end for the code generator.

[b] Currently, Apache CXF only provides the JAXB data binding for the code generator.

[c] Currently, Apache CXF only provides WSDL 1.1 support for the code generator.

NAME

java2ws — generates a WSDL document from Java code

SYNOPSIS

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>version</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <option>...</option>
         ...
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

DESCRIPTION

The **java2ws** task takes a service endpoint implementation (SEI) and generates the support files used to implement a Web service. It can generate the following:

- a WSDL document
- the server code needed to deploy the service as a POJO
- client code for accessing the service
- wrapper and fault beans

REQUIRED CONFIGURATION

The plug-in requires that the **className** configuration element is present. The element's value is the fully qualified name of the SEI to be processed.

OPTIONAL CONFIGURATION

The configuration element's listed in the following table can be used to fine tune the WSDL generation.

Element	Description
frontend	Specifies front end to use for processing the SEI and generating the support classes. **jaxws** is the default. **simple** is also supported.
databinding	Specifies the data binding used for processing the SEI and generating the support classes. The default when using the JAX-WS front end is **jaxb**. The default when using the simple frontend is **aegis**.

Element	Description
genWsdl	Instructs the tool to generate a WSDL document when set to **true**.
genWrapperbean	Instructs the tool to generate the wrapper bean and the fault beans when set to **true**.
genClient	Instructs the tool to generate client code when set to **true**.
genServer	Instructs the tool to generate server code when set to **true**.
outputFile	Specifies the name of the generated WSDL file.
classpath	Specifies the classpath searched when processing the SEI.
soap12	Specifies that the generated WSDL document is to include a SOAP 1.2 binding when set to **true**.
targetNamespace	Specifies the target namespace to use in the generated WSDL file.
serviceName	Specifies the value of the generated**service** element's **name** attribute.

PART VI. DEVELOPING RESTFUL WEB SERVICES

Abstract

This guide describes how to use the JAX-RS APIs to implement Web services.

CHAPTER 44. INTRODUCTION TO RESTFUL WEB SERVICES

Abstract

Representational State Transfer(REST) is a software architecture style that centers around the transmission of data over HTTP, using only the four basic HTTP verbs. It also eschews the use of any additional wrappers such as a SOAP envelope and the use of any state data.

OVERVIEW

Representational State Transfer(REST) is an architectural style first described in a doctoral dissertation by a researcher named Roy Fielding. In RESTful systems, servers expose resources using a URI, and clients access these resources using the four HTTP verbs. As clients receive representations of a resource they are placed in a state. When they access a new resource, typically by following a link, they change, or transition, their state. In order to work, REST assumes that resources are capable of being represented using a pervasive standard grammar.

The World Wide Web is the most ubiquitous example of a system designed on REST principles. Web browsers act as clients accessing resources hosted on Web servers. The resources are represented using HTML or XML grammars that all Web browsers can consume. The browsers can also easily follow the links to new resources.

The advantages of RESTful systems is that they are highly scalable and highly flexible. Because the resources are accessed and manipulated using the four HTTP verbs, the resources are exposed using a URIs, and the resources are represented using standard grammars, clients are not as affected by changes to the servers. Also, RESTful systems can take full advantage of the scalability features of HTTP such as caching and proxies.

BASIC REST PRINCIPLES

RESTful architectures adhere to the following basic principles:

- Application state and functionality are divided into resources.
- Resources are addressable using standard URIs that can be used as hypermedia links.
- All resources use only the four HTTP verbs.
  - **DELETE**
  - **GET**
  - **POST**
  - **PUT**
- All resources provide information using the MIME types supported by HTTP.
- The protocol is stateless.
- Responses are cacheable.
- The protocol is layered.

RESOURCES

Resources are central to REST. A resource is a source of information that can be addressed using a URI. In the early days of the Web, resources were largely static documents. In the modern Web, a resource can be any source of information. For example a Web service can be a resource if it can be accessed using a URI.

RESTful endpoints exchange *representations* of the resources they address. A representation is a document containing the data provided by the resource. For example, the method of a Web service that provides access to a customer record would be a resource, the copy of the customer record exchanged between the service and the consumer is a representation of the resource.

REST BEST PRACTICES

When designing RESTful Web services it is helpful to keep in mind the following:

- Provide a distinct URI for each resource you wish to expose.

  For example, if you are building a system that deals with driving records, each record should have a unique URI. If the system also provides information on parking violations and speeding fines, each type of resource should also have a unique base. For example, speeding fines could be accessed through /speedingfines/*driverID* and parking violations could be accessed through /parkingfines/*driverID*.

- Use nouns in your URIs.

  Using nouns highlights the fact that resources are things and not actions. URIs such as /ordering imply an action, whereas /orders implies a thing.

- Methods that map to **GET** should not change any data.

- Use links in your responses.

  Putting links to other resources in your responses makes it easier for clients to follow a chain of data. For example, if your service returns a collection of resources, it would be easier for a client to access each of the individual resources using the provided links. If links are not included, a client needs to have additional logic to follow the chain to a specific node.

- Make your service stateless.

  Requiring the client or the service to maintain state information forces a tight coupling between the two. Tight couplings make upgrading and migrating more difficult. Maintaining state can also make recovery from communication errors more difficult.

DESIGNING A RESTFUL WEB SERVICE

Regardless of the framework you use to implement a RESTful Web service, there are a number of steps that should be followed:

1. Define the resources the service will expose.

   In general, a service will expose one or more resources that are organized as a tree. For example, a driving record service could be organized into three resources:

   - /license/*driverID*

- /license/*driverID*/speedingfines
- /license/*driverID*/parkingfines

2. Define what actions you want to be able to perform on each resource.

   For example, you may want to be able to update a diver's address or remove a parking ticket from a driver's record.

3. Map the actions to the appropriate HTTP verbs.

Once you have defined the service, you can implement it using Apache CXF.

IMPLEMENTING REST WITH APACHE CXF

Apache CXF provides an implementation of the *Java API for RESTFul Web Services*(JAX-RS). JAX-RS provides a standardized way to map POJOs to resources using annotations.

When moving from the abstract service definition to a RESTful Web service implemented using JAX-RS, you need to do the following:

1. Create a root resource class for the resource that represents the top of the service's resource tree.

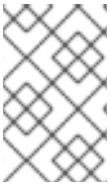
   See Section 45.3, “Root resource classes”.

2. Map the service's other resources into sub-resources.

   See Section 45.5, “Working with sub-resources”.

3. Create methods to implement each of the HTTP verbs used by each of the resources.

   See Section 45.4, “Working with resource methods”.



NOTE

Apache CXF continues to support the old HTTP binding to map Java interfaces into RESTful Web services. The HTTP binding provides basic functionality and has a number of limitations. Developers are encouraged to update their applications to use JAX-RS.

DATA BINDINGS

By default, Apache CXF uses Java Architecture for XML Binding(JAXB) objects to map the resources and their representations to Java objects. Provides clean, well defined mappings between Java objects and XML elements.

The Apache CXF JAX-RS implementation also supports exchanging data using *JavaScript Object Notation*(JSON). JSON is a popular data format used by Ajax developers. The marshaling of data between JSON and JAXB is handled by the Apache CXF runtime.

CHAPTER 45. CREATING RESOURCES

Abstract

In RESTful Web services all requests are handled by resources. The JAX-RS APIs implement resources as a Java class. A resource class is a Java class that is annotated with one, or more, RAX-RS annotations. The core of a RESTful Web service implemented using JAX-RS is a root resource class. The root resource class is the entry point to the resource tree exposed by a service. It may handle all requests itself, or it may provide access to sub-resources that handle requests.

45.1. INTRODUCTION

Overview

RESTful Web services implemented using JAX-RS APIs provide responses as representations of a resource implemented by Java classes. A *resource class* is a class that uses JAX-RS annotations to implement a resource. For most RESTful Web services, there is a collection of resources that need to be accessed. The resource class' annotations provide information such as the URI of the resources and which HTTP verb each operation handles.

Types of resources

The JAX-RS APIs allow you to create two basic types of resources:

- A Section 45.3, “Root resource classes” is the entry point to a service's resource tree. It is decorated with the **@Path** annotation to define the base URI for the resources in the service.
- Section 45.5, “Working with sub-resources” are accessed through the root resource. They are implemented by methods that are decorated with the **@Path** annotation. A sub-resource's **@Path** annotation defines a URI relative to the base URI of a root resource.

Example

Example 45.1, “Simple resource class” shows a simple resource class.

Example 45.1. Simple resource class

```
package demo.jaxrs.server;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;

(1) @Path("/customerservice")
  public class CustomerService
{
  public CustomerService()
  {
  }

(2)   @GET
      public Customer getCustomer(@QueryParam("id") String id)
```

```
    {
      ...
    }

    ...
}
```

Two items make the class defined in Example 45.1, “Simple resource class” a resource class:

1. The **@Path** annotation specifies the base URI for the resource.
2. The **@GET** annotation specifies that the method implements the HTTP **GET** method for the resource.

45.2. BASIC JAX-RS ANNOTATIONS

Overview

The most basic pieces of information required by a RESTful Web service implementation are:

- the URI of the service's resources
- how the class' methods are mapped to the HTTP verbs

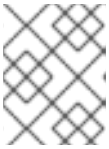
JAX-RS defines a set of annotations that provide this basic information. All resource classes must have at least one of these annotations.

Setting the path

The **@Path** annotation specifies the URI of a resource. The annotation is defined by the **javax.ws.rs.Path** interface and it can be used to decorate either a resource class or a resource method. It takes a string value as its only parameter. The string value is a URI template that specifies the location of an implemented resource.

The URI template specifies a relative location for the resource. As shown in Example 45.2, “URI template syntax”, the template can contain the following:

- unprocessed path components
- parameter identifiers surrounded by **{ }**



NOTE

Parameter identifiers can include regular expressions to alter the default path processing.

Example 45.2. URI template syntax

```
@Path("resourceName/{param1}/../{paramN}")
```

For example, the URI template widgets/{color}/{number} would map to widgets/blue/12. The value of the ***color*** parameter is assigned to blue. The value of the ***number*** parameter is assigned 12.

How the URI template is mapped to a complete URI depends on what the **@Path** annotation is decorating. If it is placed on a root resource class, the URI template is the root URI of all resources in the tree and it is appended directly to the URI at which the service is published. If the annotation decorates a sub-resource, it is relative to the root resource URI.

Specifying HTTP verbs

JAX-RS uses five annotations for specifying the HTTP verb that will be used for a method:

- **javax.ws.rs.DELETE** specifies that the method maps to a **DELETE**.
- **javax.ws.rs.GET** specifies that the method maps to a **GET**.
- **javax.ws.rs.POST** specifies that the method maps to a **POST**.
- **javax.ws.rs.PUT** specifies that the method maps to a **PUT**.
- **javax.ws.rs.HEAD** specifies that the method maps to a **HEAD**.

When you map your methods to HTTP verbs, you must ensure that the mapping makes sense. For example, if you map a method that is intended to submit a purchase order, you would map it to a **PUT** or a **POST**. Mapping it to a **GET** or a **DELETE** would result in unpredictable behavior.

45.3. ROOT RESOURCE CLASSES

Overview

A root resource class is the entry point into a JAX-RS implemented RESTful Web service. It is decorated with a **@Path** that specifies the root URI of the resources implemented by the service. Its methods either directly implement operations on the resource or provide access to sub-resources.

Requirements

In order for a class to be a root resource class it must meet the following criteria:

- The class must be decorated with the **@Path** annotation.

  The specified path is the root URI for all of the resources implemented by the service. If the root resource class specifies that its path is widgets and one of its methods implements the **GET** verb, then a **GET** on widgets invokes that method. If a sub-resource specifies that its URI is {id}, then the full URI template for the sub-resource is widgets/{id} and it will handle requests made to URIs like widgets/12 and widgets/42.

- The class must have a public constructor for the runtime to invoke.

  The runtime must be able to provide values for all of the constructor's parameters. The constructor's parameters can include parameters decorated with the JAX-RS parameter annotations. For more information on the parameter annotations see Chapter 46, *Passing Information into Resource Classes and Methods*.

- At least one of the classes methods must either be decorated with an HTTP verb annotation or the **@Path** annotation.

Example

Example 45.3, "Root resource class" shows a root resource class that provides access to a sub-resource.

Example 45.3. Root resource class

```
package demo.jaxrs.server;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

(1) @Path("/customerservice/")
  public class CustomerService
{
(2)   public CustomerService()
    {
    ...
  }

(3)   @GET
    public Customer getCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @DELETE
  public Response deleteCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @PUT
  public Response updateCustomer(Customer customer)
  {
    ...
  }

  @POST
  public Response addCustomer(Customer customer)
  {
    ...
  }

(4)   @Path("/orders/{orderId}/")
    public Order getOrder(@PathParam("orderId") String orderId)
```

```
    {
      ...
    }

}
```

The class in Example 45.3, "Root resource class" meets all of the requirements for a root resource class.

1. The class is decorated with the **@Path** annotation. The root URI for the resources exposed by the service is customerservice.
2. The class has a public constructor. In this case the no argument constructor is used for simplicity.
3. The class implements each of the four HTTP verbs for the resource.
4. The class also provides access to a sub-resource through the **getOrder()** method. The URI for the sub-resource, as specified using the the **@Path** annotation, is customerservice/order/*id*. The sub-resource is implemented by the **Order** class.

   For more information on implementing sub-resources see Section 45.5, "Working with sub-resources".

45.4. WORKING WITH RESOURCE METHODS

Overview

Resource methods are annotated using JAX-RS annotations. They have one of the HTTP method annotation specifying the types of requests that the method processes. JAX-RS places several constraints on resource methods.

General constraints

All resource methods must meet the following conditions:

- It must be public.
- It must be decorated with one of the HTTP method annotations described in the section called "Specifying HTTP verbs".
- It must not have more than one entity parameter as described in the section called "Parameters".

Parameters

Resource method parameters take two forms:

- **entity parameters**—Entity parameters are not annotated. Their value is mapped from the request entity body. An entity parameter can be of any type for which your application has an entity provider. Typically they are JAXB objects.

IMPORTANT

A resource method can have *only one* entity parameter.

For more information on entity providers see Chapter 49, *Entity Support*.

- **annotated parameters**—Annotated parameters use one of the JAX-RS annotations that specify how the value of the parameter is mapped from the request. Typically, the value of the parameter is mapped from portions of the request URI.

  For more information about using the JAX-RS annotations for mapping request data to method parameters see Chapter 46, *Passing Information into Resource Classes and Methods*

Example 45.4, "Resource method with a valid parameter list" shows a resource method with a valid parameter list.

Example 45.4. Resource method with a valid parameter list

```
@POST
@Path("disaster/monster/giant/{id}")
public void addDaikaiju(Kaiju kaiju,
                        @PathParam("id") String id)
{
  ...
}
```

Example 45.5, "Resource method with an invalid parameter list" shows a resource method with an invalid parameter list. It has two parameters that are not annotated.

Example 45.5. Resource method with an invalid parameter list

```
@POST
@Path("disaster/monster/giant/")
public void addDaikaiju(Kaiju kaiju,
                        String id)
{
  ...
}
```

Return values

Resource methods can return one of the following:

- **`void`**
- any Java class for which the application has an entity provider

  For more information on entity providers see Chapter 49, *Entity Support*.

- a **`Response`** object

For more information on **Response** objects see Section 47.2, "Fine tuning an application's responses".

- a **GenericEntity<*T*>** object

  For more information on **GenericEntity<*T*>** objects see Section 47.3, "Returning entities with generic type information".

All resource methods return an HTTP status code to the requester. When the return type of the method is **void** or the value being returned is **null**, the resource method sets the HTTP status code to **204**. When the resource method returns any value other than **null**, it sets the HTTP status code to **200**.

45.5. WORKING WITH SUB-RESOURCES

Overview

It is likely that a service will need to be handled by more than one resource. For example, in an order processing service best-practices suggests that each customer would be handled as a unique resource. Each order would also be handled as a unique resource.

Using the JAX-RS APIs, you would implement the customer resources and the order resources as *sub-resources*. A sub-resource is a resource that is accessed through a root resource class. They are defined by adding a **@Path** annotation to a resource class' method. Sub-resources can be implemented in one of two ways:

- *Sub-resource method*—directly implements an HTTP verb for a sub-resource and is decorated with one of the annotations described in the section called "Specifying HTTP verbs" .
- *Sub-resource locator*—points to a class that implements the sub-resource.

Specifying a sub-resource

Sub-resources are specified by decorating a method with the **@Path** annotation. The URI of the sub-resource is constructed as follows:

1. Append the value of the sub-resource's **@Path** annotation to the value of the sub-resource's parent resource's **@Path** annotation.

   The parent resource's **@Path** annotation maybe located on a method in a resource class that returns an object of the class containing the sub-resource.

2. Repeat the previous step until the root resource is reached.

3. The assembled URI is appended to the base URI at which the service is deployed.

For example the URI of the sub-resource shown in Example 45.6, "Order sub-resource" could be *baseURI*/customerservice/order/12.

Example 45.6. Order sub-resource

```
...
@Path("/customerservice/")
public class CustomerService
```

```
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }
}
```

Sub-resource methods

A sub-resource method is decorated with both a **@Path** annotation and one of the HTTP verb annotations. The sub-resource method is directly responsible for handling a request made on the resource using the specified HTTP verb.

Example 45.7, "Sub-resource methods" shows a resource class with three sub-resource methods:

- **getOrder()** handles HTTP **GET** requests for resources whose URI matches /customerservice/orders/{orderId}/.
- **updateOrder()** handles HTTP **PUT** requests for resources whose URI matches /customerservice/orders/{orderId}/.
- **newOrder()** handles HTTP **POST** requests for the resource at /customerservice/orders/.

Example 45.7. Sub-resource methods

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @Path("/orders/{orderId}/")
  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
  {
    ...
  }

  @Path("/orders/")
  @POST
  public Order newOrder(Order order)
  {
```

```
    ...
  }
}
```

NOTE

Sub-resource methods with the same URI template are equivalent to resource class returned by a sub-resource locator.

Sub-resource locators

Sub-resource locators are not decorated with one of the HTTP verb annotations and do not directly handle are request on the sub-resource. Instead, a sub-resource locator returns an instance of a resource class that can handle the request.

In addition to not having an HTTP verb annotation, sub-resource locators also cannot have any entity parameters. All of the parameters used by a sub-resource locator method must use one of the annotations described in Chapter 46, *Passing Information into Resource Classes and Methods*

As shown in Example 45.8, "Sub-resource locator returning a specific class" , sub-resource locator allows you to encapsulate a resource as a reusable class instead of putting all of the methods into one super class. The **`processOrder()`** method is a sub-resource locator. When a request is made on a URI matching the URI template /orders/{orderId}/ it returns an instance of the **`Order`** class. The **`Order`** class has methods that are decorated with HTTP verb annotations. A **PUT** request is handled by the **`updateOrder()`** method.

Example 45.8. Sub-resource locator returning a specific class

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  public Order processOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  ...
}

public class Order
{
  ...
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
```

```
                                Order order)
  {
    ...
  }

}
```

Sub-resource locators are processed at runtime so that they can support polymorphism. The return value of a sub-resource locator can be a generic **`Object`**, an abstract class, or the top of a class hierarchy. For example, if your service needed to process both PayPal orders and credit card orders, the **`processOrder()`** method's signature from Example 45.8, “Sub-resource locator returning a specific class” could remain unchanged. You would simply need to implement two classes, **`ppOrder`** and **`ccOder`**, that extended the **`Order`** class. The implementation of **`processOrder()`** would instantiate the desired implementation of the sub-resource based on what ever logic is required.

45.6. RESOURCE SELECTION METHOD

Overview

It is possible for a given URI to map to one or more resource methods. For example the URI customerservice/12/ma could match the templates **`@Path("customerservice/{id}")`** or **`@Path("customerservice/{id}/{state}")`**. JAX-RS specifies a detailed algorithm for matching a resource method to a request. The algorithm compares the normalized URI, the HTTP verb, and the media types of the request and response entities to the annotations on the resource classes.

The basic selection algorithm

The JAX-RS selection algorithm is broken down into three stages:

1. Determine the root resource class.

   The request URI is matched against all of the classes decorated with the **`@Path`** annotation. The classes whose **`@Path`** annotation matches the request URI are determined.

   If the value of the resource class' **`@Path`** annotation matches the entire request URI, the class' methods are used as input into the third stage.

2. Determine the object will handle the request.

   If the request URI is longer than the value of the selected class' **`@Path`** annotation, the values of the resource methods' **`@Path`** annotations are used to look for a sub-resource that can process the request.

   If one or more sub-resource methods match the request URI, these methods are used as input for the third stage.

   If the only matches for the request URI are sub-resource locaters, the resource methods of the object created by the sub-resource locater to match the request URI. This stage is repeated until a sub-resource method matches the request URI.

3. Select the resource method that will handle the request.

   The resource method whose HTTP verb annotation matches the HTTP verb in the request. In

addition, the selected resource method must accept the media type of the request entity body and be capable of producing a response that conforms to the media type(s) specified in the request.

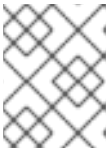
Selecting from multiple resource classes

The first two stages of the selection algorithm determine the resource that will handle the request. In some cases the resource is implemented by a resource class. In other cases, it is implemented by one or more sub-resources that use the same URI template. When there are multiple resources that match a request URI, resource classes are preferred over sub-resources.

If more than one resource still matches the request URI after sorting between resource classes and sub-resources, the following criteria are used to select a single resource:

1. Prefer the resource with the most literal characters in its URI template.

   Literal characters are characters that are not part of a template variable. For example, /widgets/{id}/{color} has ten literal characters and /widgets/1/{color} has eleven literal characters. So, the request URI /widgets/1/red would be matched to the resource with /widgets/1/{color} as its URI template.



NOTE

A trailing slash (/) counts as a literal character. So /joefred/ will be preferred over /joefred.

2. Prefer the resource with the most variables in its URI template.

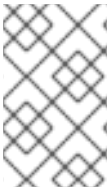
   The request URI /widgets/30/green could match both /widgets/{id}/{color} and /widgets/{amount}/. However, the resource with the URI template /widgets/{id}/{color} will be selected because it has two variables.

3. Prefer the resource with the most variables containing regular expressions.

   The request URI /widgets/30/green could match both /widgets/{number}/{color} and /widgets/{id:.+}/{color}. However, the resource with the URI template /widgets/{id:.+}/{color} will be selected because it has a variable containing a regular expression.

Selecting from multiple resource methods

In many cases, selecting a resource that matches the request URI results in a single resource method that can process the request. The method is determined by matching the HTTP verb specified in the request with a resource method's HTTP verb annotation. In addition to having the appropriate HTTP verb annotation, the selected method must also be able to handle the request entity included in the request and be able to produce the proper type of response specified in the request's metadata.



NOTE

The type of request entity a resource method can handle is specified by the **@Consumes** annotation. The type of responses a resource method can produce are specified using the **@Produces** annotation.

When selecting a resource produces multiple methods that can handle a request the following criteria is used to select the resource method that will handle the request:

1. Prefer resource methods over sub-resources.
2. Prefer sub-resource methods over sub-resource locaters.
3. Prefer methods that use the most specific values in the **@Consumes** annotation and the **@Produces** annotation.

   For example, a method that has the annotation **@Consumes(text/xml)** would be preferred over a method that has the annotation **@Consumes(text/*)**. Both methods would be preferred over a method without an **@Consumes** annotation or the annotation **@Consumes(*/*)**.
4. Prefer methods that most closely match the content type of the request body entity.

   TIP

   The content type of the request body entity is specified in the HTTP Content-Type property.
5. Prefer methods that most closely match the content type accepted as a response.

   TIP

   The content types accepted as a response are specified in the HTTP Accept property.

Customizing the selection process

In some cases, developers have reported the algorithm being somewhat restrictive in the way multiple resource classes are selected. For example, if a given resource class has been matched and if this class has no matching resource method, then the algorithm stops executing. It never checks the remaining matching resource classes.

Apache CXF provides the **org.apache.cxf.jaxrs.ext.ResourceComparator** interface which can be used to customize how the runtime handles multiple matching resource classes. The **ResourceComparator** interface, shown in Example 45.9, "Interface for customizing resource selection", has to methods that need to be implemented. One compares two resource classes and the other compares two resource methods.

Example 45.9. Interface for customizing resource selection

```
package org.apache.cxf.jaxrs.ext;

import org.apache.cxf.jaxrs.model.ClassResourceInfo;
import org.apache.cxf.jaxrs.model.OperationResourceInfo;
import org.apache.cxf.message.Message;

public interface ResourceComparator
{
  int compare(ClassResourceInfo cri1,
              ClassResourceInfo cri2,
              Message message);

  int compare(OperationResourceInfo oper1,
```

```
                OperationResourceInfo oper2,
                Message message);
    }
```

Custom implementations select between the two resources as follows:

- Return **1** if the first parameter is a better match than the second parameter
- Return **-1** if the second parameter is a better match than the first parameter

If **0** is returned then the runtime will proceed with the default selection algorithm

You register a custom **ResourceComparator** implementation by adding a **resourceComparator** child to the service's **jaxrs:server** element.

CHAPTER 46. PASSING INFORMATION INTO RESOURCE CLASSES AND METHODS

Abstract

JAX-RS specifies a number of annotations that allow the developer to control where the information passed into resources come from. The annotations conform to common HTTP concepts such as matrix parameters in a URI. The standard APIs allow the annotations to be used on method parameters, bean properties, and resource class fields. Apache CXF provides an extension that allows for the injection of a sequence of parameters to be injected into a bean.

46.1. BASICS OF INJECTING DATA

Overview

Parameters, fields, and bean properties that are initialized using data from the HTTP request message have their values injected into them by the runtime. The specific data that is injected is specified by a set of annotations described in Section 46.2, "Using JAX-RS APIs".

The JAX-RS specification places a few restrictions on when the data is injected. It also places a few restrictions on the types of objects into which request data can be injected.

When data is injected

Request data is injected into objects when they are instantiated due to a request. This means that only objects that directly correspond to a resource can use the injection annotations. As discussed in Chapter 45, *Creating Resources*, these objects will either be a root resource decorated with the `@Path` annotation or an object returned from a sub-resource locator method.

Supported data types

The specific set of data types that data can be injected into depends on the annotation used to specify the source of the injected data. However, all of the injection annotations support at least the following set of data types:

- primitives such as int, char, or long
- Objects that have a constructor that accepts a single String argument
- Objects that have a static `valueOf()` method that accepts a single String argument
- `List<T>`, `Set<T>`, or `SortedSet<T>` objects where *T* satisfies the other conditions in the list

TIP

Where injection annotations have different requirements for supported data types, the differences will be highlighted in the discussion of the annotation.

46.2. USING JAX-RS APIS

The standard JAX-RS API specifies annotations that can be used to inject values into fields, bean properties, and method parameters. The annotations can be split up into three distinct types:

- annotations that inject information from the request URI
- annotations that inject information from the HTTP message header
- annotations that inject information from HTML forms

46.2.1. Injecting data from a request URI

Overview

One of the best practices for designing a RESTful Web service is that each resource should have a unique URI. A developer can use this principle to provide a good deal of information to the underlying resource implementation. When designing URI templates for a resource, a developer can build the templates to include parameter information that can be injected into the resource implementation. Developers can also leverage query and matrix parameters for feeding information into the resource implementations.

Getting data from the URI's path

One of the more common mechanisms for getting information about a resource is through the variables used in creating the URI templates for a resource. This is accomplished using the **`javax.ws.rs.PathParam`** annotation. The **`@PathParam`** annotation has a single parameter that identifies the URI template variable from which the data will be injected.

In Example 46.1, "Injecting data from a URI template variable" the **`@PathParam`** annotation specifies that the value of the URI template variable **`color`** is injected into the **`itemColor`** field.

Example 46.1. Injecting data from a URI template variable

```
import javax.ws.rs.Path;
import javax.ws.rs.PathParam
...

@Path("/boxes/{shape}/{color}")
class Box
{
  ...

  @PathParam("color")
  String itemColor;

  ...
}
```

The data types supported by the **`@PathParam`** annotation are different from the ones described in the section called "Supported data types". The entity into which the **`@PathParam`** annotation injects data must be of one of the following types:

- **`PathSegment`**

The value will be the final segment of the matching part of the path.

- **`List<PathSegment>`**

  The value will be a list of **`PathSegment`** objects corresponding to the path segment(s) that matched the named template parameter.

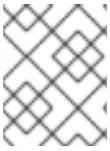
- primitives such as int, char, or long
- Objects that have a constructor that accepts a single String argument
- Objects that have a static **`valueOf()`** method that accepts a single String argument

Using query parameters

A common way of passing information on the Web is to use *query parameters* in a URI. Query parameters appear at the end of the URI and are separated from the resource location portion of the URI by a question mark(**?**). They consist of one, or more, name value pairs where the name and value are separated by an equal sign(**=**). When more than one query parameter is specified, the pairs are separated from each other by either a semicolon(**;**) or an ampersand(**&**). Example 46.2, "URI with a query string" shows the syntax of a URI with query parameters.

Example 46.2. URI with a query string

```
http://fusesource.org?name=value;name2=value2;...
```



NOTE

You can use **either** the semicolon or the ampersand to separate query parameters, but not both.

The **`javax.ws.rs.QueryParam`** annotation extracts the value of a query parameter and injects it into a JAX-RS resource. The annotation takes a single parameter that identifies the name of the query parameter from which the value is extracted and injected into the specified field, bean property, or parameter. The **`@QueryParam`** annotation supports the types described in the section called "Supported data types".

Example 46.3, "Resource method using data from a query parameter" shows a resource method that injects the value of the query parameter ***`id`*** into the method's ***`id`*** parameter.

Example 46.3. Resource method using data from a query parameter

```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
```

```
  @Path("\{type}")
  public void updateMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
    ...
  }
  ...
}
```

To process an HTTP **POST** to /monstersforhire/daikaiju?id=jonas the **updateMonster()** method's ***type*** is set to **daikaiju** and the ***id*** is set to **jonas**.

Using matrix parameters

URI matrix parameters, like URI query parameters, are name/value pairs that can provide additional information selecting a resource. Unlike query parameters, matrix parameters can appear anywhere in a URI and they are separated from the hierarchical path segments of the URI using a semicolon(;). /mostersforhire/daikaiju;id=jonas has one matrix parameter called ***id*** and /monstersforhire/japan;type=daikaiju/flying;wingspan=40 has two matrix parameters called ***type*** and ***wingspan***.

NOTE

Matrix parameters are not evaluated when computing a resource's URI. So, the URI used to locate the proper resource to handle the request URI /monstersforhire/japan;type=daikaiju/flying;wingspan=40 is /monstersforhire/japan/flying.

The value of a matrix parameter is injected into a field, parameter, or bean property using the **javax.ws.rs.MatrixParam** annotation. The annotation takes a single parameter that identifies the name of the matrix parameter from which the value is extracted and injected into the specified field, bean property, or parameter. The **@MatrixParam** annotation supports the types described in the section called “Supported data types”.

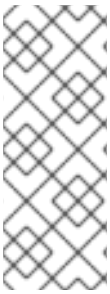
Example 46.4, “Resource method using data from matrix parameters” shows a resource method that injects the value of the matrix parameters ***type*** and ***id*** into the method's parameters.

Example 46.4. Resource method using data from matrix parameters

```
import javax.ws.rs.MatrixParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  public void updateMonster(@MatrixParam("type") String type,
                            @MatrixParam("id") String id)
  {
    ...
```

```
    }
    ...
}
```

To process an HTTP **POST** to /monstersforhire;type=daikaiju;id=whale the **updateMonster()** method's ***type*** is set to **daikaiju** and the ***id*** is set to **whale**.



NOTE

JAX-RS evaluates all of the matrix parameters in a URI at once, so it cannot enforce constraints on a matrix parameters location in a URI. For example /monstersforhire/japan;type=daikaiju/flying;wingspan=40 , /monstersforhire/japan/flying;type=daikaiju;wingspan=40, and /monstersforhire/japan;type=daikaiju;wingspan=40/flying are all treated as equivalent by a RESTful Web service implemented using the JAX-RS APIs.

Disabling URI decoding

By default all request URIs are decoded. So the URI /monster/night%20stalker and the URI /monster/night stalker are equivalent. The automatic URI decoding makes it easy to send characters outside of the ASCII character set as parameters.

If you do not wish to have URI automatically decoded, you can use the **javax.ws.rs.Encoded** annotation to deactivate the URI decoding. The annotation can be used to deactivate URI decoding at the following levels:

- class level—Decorating a class with the **@Encoded** annotation deactivates the URI decoding for all parameters, field, and bean properties in the class.
- method level—Decorating a method with the **@Encoded** annotation deactivates the URI decoding for all parameters of the class.
- parameter/field level—Decorating a parameter or field with the **@Encoded** annotation deactivates the URI decoding for all parameters of the class.

Example 46.5, “Disabling URI decoding” shows a resource whose **getMonster()** method does not use URI decoding. The **addMonster()** method only disables URI decoding for the ***type*** parameter.

Example 46.5. Disabling URI decoding

```
@Path("/monstersforhire/")
public class MonsterService
{
  ...

  @GET
  @Encoded
  @Path("\{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
    ...
  }
```

```
  @PUT
  @Path("\{id}")
  public void addMonster(@Encoded @PathParam("type") String type,
                         @QueryParam("id") String id)
  {
    ...
  }
  ...
}
```

Error handling

If an error occurs when attempting to inject data using one of the URI injection annotations a **`WebApplicationException`** exception wraps the original exception is generated. The **`WebApplicationException`** exception's status is set to **`404`**.

46.2.2. Injecting data from the HTTP message header

Overview

In normal usage the HTTP headers in a request message pass along generic information about the message, how it is to be handled in transit, and details about the expected response. While a few standard headers are commonly recognized and used, the HTTP specification allows for any name/value pair to be used as an HTTP header. The JAX-RS APIs provide an easy mechanism for injecting HTTP header information into a resource implementation.

One of the most commonly used HTTP headers is the cookie. Cookies allow HTTP clients and servers to share static information across multiple request/response sequences. The JAX-RS APIs provide an annotation inject data directly from a cookie into a resource implementation.

Injecting information from the HTTP headers

The **`javax.ws.rs.HeaderParam`** annotation is used to inject the data from an HTTP header field into a parameter, field, or bean property. It has a single parameter that specifies the name of the HTTP header field from which the value is extracted and injected into the resource implementation. The associated parameter, field, or bean property must conform to the data types described in the section called “Supported data types”.

Example 46.6, “Injecting the If-Modified-Since header” shows code for injecting the value of the HTTP **`If-Modified-Since`** header into a class' **`oldestDate`** field.

Example 46.6. Injecting the If-Modified-Since header

```
import javax.ws.rs.HeaderParam;
...
class RecordKeeper
{
  ...
  @HeaderParam("If-Modified-Since")
  String oldestDate;
  ...
}
```

Injecting information from a cookie

Cookies are a special type of HTTP header. They are made up of one or more name/value pairs that are passed to the resource implementation on the first request. After the first request, the cookie is passes back and forth between the provider and consumer with each message. Only the consumer, because they generate requests, can change the cookie. Cookies are commonly used to maintain session across multiple request/response sequences, storing user settings, and other data that can persist.

The **javax.ws.rs.CookieParam** annotation extracts the value from a cookie's field and injects it into a resource implementation. It takes a single parameter that specifies the name of the cookie's field from which the value is to be extracted. In addition to the data types listed in the section called "Supported data types", entities decorated with the **@CookieParam** can also be a **Cookie** object.

Example 46.7, "Injecting a cookie" shows code for injecting the value of the **handle** cookie into a field in the **CB** class.

Example 46.7. Injecting a cookie

```
import javax.ws.rs.CookieParam;
...
class CB
{
  ...
  @CookieParam("handle")
  String handle;
  ...
}
```

Error handling

If an error occurs when attempting to inject data using one of the HTTP message injection annotations a **WebApplicationException** exception wrapping the original exception is generated. The **WebApplicationException** exception's status is set to **400**.

46.2.3. Injecting data from HTML forms

Overview

HTML forms are an easy means of getting information from a user and they are also easy to create. Form data can be used for HTTP **GET** requests and HTTP **POST** requests:

GET

When form data is sent as part of an HTTP **GET** request the data is appended to the URI as a set of query parameters. Injecting data from query parameters is discussed in the section called "Using query parameters".

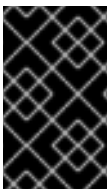
POST

When form data is sent as part of an HTTP **POST** request the data is placed in the HTTP message

body. The form data can be handled using a regular entity parameter that supports the form data. It can also be handled by using the **@FormParam** annotation to extract the data and inject the pieces into resource method parameters.

Using the @FormParam annotation to inject form data

The **javax.ws.rs.FormParam** annotation extracts field values from form data and injects the value into resource method parameters. The annotation takes a single parameter that specifies the key of the field from which it extracts the values. The associated parameter must conform to the data types described in the section called "Supported data types".



IMPORTANT

The JAX-RS API Javadoc states that the **@FormParam** annotation can be placed on fields, methods, and parameters. However, the **@FormParam** annotation is only meaningful when placed on resource method parameters.

Example

Example 46.8, "Injecting form data into resource method parameters" shows a resource method that injects form data into its parameters. The method assumes that the client's form includes three fields —**title**, **tags**, and **body**—that contain string data.

Example 46.8. Injecting form data into resource method parameters

```
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;

...
@POST
public boolean updatePost(@FormParam("title") String title,
                          @FormParam("tags") String tags,
                          @FormParam("body") String post)
{
  ...
}
```

46.2.4. Specifying a default value to inject

Overview

To provide for a more robust service implementation, you may want to ensure that any optional parameters can be set to a default value. This can be particularly useful for values that are taken from query parameters and matrix parameters since entering long URI strings is highly error prone. You may also want to set a default value for a parameter extracted from a cookie since it is possible for a requesting system not have the proper information to construct a cookie with all the values.

The **javax.ws.rs.DefaultValue** annotation can be used in conjunction with the following injection annotations:

- **@PathParam**

- **@QueryParam**
- **@MatrixParam**
- **@FormParam**
- **@HeaderParam**
- **@CookieParam**

The **@DefaultValue** annotation specifies a default value to be used when the data corresponding to the injection annotation is not present in the request.

Syntax

Example 46.9, "Syntax for setting the default value of a parameter" shows the syntax for using the **@DefaultValue** annotation.

Example 46.9. Syntax for setting the default value of a parameter

```
import javax.ws.rs.DefaultValue;
  ...
  void resourceMethod(@MatrixParam("matrix")
                      @DefaultValue("value)
                      int someValue, ... )
  ...
```

The annotation must come before the parameter, bean, or field, it will effect. The position of the **@DefaultValue** annotation relative to the accompanying injection annotation does not matter.

The **@DefaultValue** annotation takes a single parameter. This parameter is the value that will be injected into the field if the proper data cannot be extracted based on the injection annotation. The value can be any String value. The value should be compatible with type of the associated field. For example, if the associated field is of type int, a default value of **blue** results in an exception.

Dealing with lists and sets

If the type of the annotated parameter, bean or field is **List**, **Set**, or **SortedSet** then the resulting collection will have a single entry mapped from the supplied default value.

Example

Example 46.10, "Setting default values" shows two examples of using the **@DefaultValue** to specify a default value for a field whose value is injected.

Example 46.10. Setting default values

```
import javax.ws.rs.DefaultValue;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
```

```
@Path("/monster")
public class MonsterService
{

  @Get
  public Monster getMonster(@QueryParam("id") @DefaultValue("42") int
id,
                            @QueryParam("type")
@DefaultValue("bogeyman") String type)
  {
    ...
  }

  ...
}
```

The **getMonster()** method in Example 46.10, "Setting default values" is invoked when a **GET** request is sent to *baseURI*/monster. The method expects two query parameters, ***id*** and ***type***, appended to the URI. So a **GET** request using the URI *baseURI*/monster?id=1&type=fomóiri would return the Fomóiri with the id of one.

Because the **@DefaultValue** annotation is placed on both parameters, the **getMonster()** method can function if the query parameters are omitted. A **GET** request sent to *baseURI*/monster is equivalent to a **GET** request using the URI *baseURI*/monster?id=42&type=bogeyman.

46.3. USING APACHE CXF EXTENSIONS

Overview

Apache CXF provides an extension to the standard JAX-WS injection mechanism that allows developers to replace a sequence of injection annotations with a single annotation. The single annotation is place on a bean containing fields for the data that is extracted using the annotation. For example, if a resource method is expecting a request URI to include three query parameters called ***id***, ***type***, and ***size***, it could use a single **@QueryParam** annotation to inject all of the parameters into a bean with corresponding fields.

Supported injection annotations

This extension does not support all of the injection parameters. It only supports the following ones:

- **@PathParam**
- **@QueryParam**
- **@MatrixParam**
- **@FormParam**

Syntax

To indicate that an annotation is going to use serial injection into a bean, you need to do two things:

1. Specify the annotation's parameter as an empty string. For example **@PathParam("")** specifies that a sequence of URI template variables are to be serialized into a bean.

2. Ensure that the annotated parameter is a bean with fields that match the values being injected.

Example

Example 46.11, "Injecting query parameters into a bean" shows an example of injecting a number of Query parameters into a bean. The resource method expect the request URI to include two query parameters: ***type*** and ***id***. Their values are injected into the corresponding fields of the **Monster** bean.

Example 46.11. Injecting query parameters into a bean

```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  public void updateMonster(@QueryParam("") Monster bean)
  {
    ...
  }
  ...
}

public class Monster
{
  String type;
  String id;

  ...
}
```

CHAPTER 47. RETURNING INFORMATION TO THE CONSUMER

Abstract

RESTful requests require that at least an HTTP response code be returned to the consumer. In many cases, a request can be satisfied by returning a plain JAXB object or a `GenericEntity` object. When the resource method needs to return additional metadata along with the response entity, JAX-RS resource methods can return a **Response** object containing any needed HTTP headers or other metadata.

The information returned to the consumer determines the exact type of object a resource method returns. This may seem obvious, but the mapping between Java return objects and what is returned to a RESTful consumer is not one-to-one. At a minimum, RESTful consumers need to be returned a valid HTTP return code in addition to any response entity body. The mapping of the data contained within a Java object to a response entity is effected by the MIME types a consumer is willing to accept.

To address the issues involved in mapping Java object to RESTful response messages, resource methods are allowed to return four types of Java constructs:

- common Java types return basic information with HTTP return codes determined by the JAX-RS runtime.
- JAXB objects return complex information with HTTP return codes determined by the JAX-RS runtime.
- JAX-RS return complex information with a programmatically determined HTTP return status. The **Response** object also allows HTTP headers to be specified.
- JAX-RS return complex information with HTTP return codes determined by the JAX-RS runtime. The `GenericEnitity` object provides more information to the runtime components serializing the data.

47.1. RETURNING PLAIN JAVA CONSTRUCTS

Overview

In many cases a resource class can return a standard Java type, a JAXB object, or any object for which the application has an entity provider. In these cases the runtime determines the MIME type information using the Java class of the object being returned. The runtime also determines the appropriate HTTP return code to send to the consumer.

Returnable types

Resource methods can return `void` or any Java type for which an entity writer is provided. By default, the runtime has providers for the following:

- the Java primitives
- the `Number` representations of the Java primitives
- JAXB objects

the section called "Natively supported types" lists all of the return types supported by default. the section called "Custom writers" describes how to implement a custom entity writer.

MIME types

The runtime determines the MIME type of the returned entity by first checking the resource method and resource class for a **@Produces** annotation. If it finds one, it uses the MIME type specified in the annotation. If it does not find one specified by the resource implementation, it relies on the entity providers to determine the proper MIME type.

By default the runtime assign MIME types as follows:

- Java primitives and their **Number** representations are assigned a MIME type of **application/octet-stream**.
- JAXB objects are assigned a MIME type of **application/xml**.

Applications can use other mappings by implementing custom entity providers as described in the section called "Custom writers".

Response codes

When resource methods return plain Java constructs, the runtime automatically sets the response's status code if the resource method completes without throwing an exception. The status code is set as follows:

- **204**(No Content)—the resource method's return type is **void**
- **204**(No Content)—the value of the returned entity is **null**
- **200**(OK)—the value of the returned entity is not **null**

If an exception is thrown before the resource method completes the return status code is set as described in Chapter 48, *Handling Exceptions*.

47.2. FINE TUNING AN APPLICATION'S RESPONSES

47.2.1. Basics of building responses

Overview

RESTful services often need more precise control over the response returned to a consumer than is allowed when a resource method returns a plain Java construct. The JAX-RS **Response** class allows a resource method to have some control over the return status sent to the consumer and to specify HTTP message headers and cookies in the response.

Response objects wrap the object representing the entity that is returned to the consumer. **Response** objects are instantiated using the **ResponseBuilder** class as a factory.

The **ResponseBuilder** class also has many of the methods used to manipulate the response's metadata. For instance the **ResonseBuilder** class contains the methods for setting HTTP headers and cache control directives.

Relationship between a response and a response builder

The **Response** class has a protected constructor, so they cannot be instantiated directly. They are created using the **ResponseBuilder** class enclosed by the **Response** class. The **ResponseBuilder** class is a holder for all of the information that will be encapsulated in the response created from it. The **ResponseBuilder** class also has all of the methods responsible for setting HTTP header properties on the message.

The **Response** class does provide some methods that ease setting the proper response code and wrapping the entity. There are methods for each of the common response status codes. The methods corresponding to status that include an entity body, or required metadata, include versions that allow for directly setting the information into the associated response builder.

The **ResponseBuilder** class' **build()** method returns a response object containing the information stored in the response builder at the time the method is invoked. After the response object is returned, the response builder is returned to a clean state.

Getting a response builder

There are two ways to get a response builder:

- Using the static methods of the **Response** class as shown in Example 47.1, "Getting a response builder using the **Response** class".

  Example 47.1. Getting a response builder using the Response class

  ```
  import javax.ws.rs.core.Response;

  Response r = Response.ok().build();
  ```

  When getting a response builder this way you do not get access to an instance you can manipulate in multiple steps. You must string all of the actions into a single method call.

- Using the Apache CXF specific **ResponseBuilderImpl** class. This class allows you to work directly with a response builder. However, it requires that you manually set all of the response builders information manually.

  Example 47.2, "Getting a response builder using the **ResponseBuilderImpl** class" shows how Example 47.1, "Getting a response builder using the **Response** class" could be rewritten using the **ResponseBuilderImpl** class.

  Example 47.2. Getting a response builder using the ResponseBuilderImpl class

  ```
  import javax.ws.rs.core.Response;
  import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

  ResponseBuilderImpl builder = new ResponseBuilderImpl();
  builder.status(200);
  Response r = builder.build();
  ```

TIP

You could also simply assign the **ResponseBuilder** returned from a **Response** class' method to a **ResponseBuilderImpl** object.

More information

For more information about the **Response** class see the **Response** class' Javadoc.

For more information about the **ResponseBuilder** class see the **ResponseBuilder** class' Javadoc.

For more information on the Apache CXF **ResponseBuilderIml** class see the **ResponseBuilderImpl** Javadoc.

47.2.2. Creating responses for common use cases

Overview

The **Response** class provides shortcut methods for handling the more common responses that a RESTful service will need. These methods handle setting the proper headers using either provided values or default values. They also handle populating the entity body when appropriate.

Creating responses for successful requests

When a request is successfully processed the application needs to send a response to acknowledge that the request has been fulfilled. That response may contain an entity.

The most common response when successfully completing a response is **OK**. An **OK** response typically contains an entity that corresponds to the request. The **Response** class has an overloaded **ok()** method that sets the response status to **200** and adds a supplied entity to the enclosed response builder. There are five versions of the **ok()** method. The most commonly used variant are:

- **Response.ok()**—creates a response with a status of **200** and an empty entity body.
- **Response.ok(java.lang.Object entity)**—creates a response with a status of **200**, stores the supplied object in the responses entity body, and determines the entities media type by introspecting the object.

Example 47.3, "Creating a response with an **200** response" shows an example of creating a response with an **OK** status.

Example 47.3. Creating a response with an 200 response

```
import javax.ws.rs.core.Response;
import demo.jaxrs.server.Customer;
...

Customer customer = new Customer("Jane", 12);

return Response.ok(customer).build();
```

For cases where the requester is not expecting an entity body, it may be more appropriate to send a **`204 No Content`** status instead of an **`200 OK`** status. The **`Response.noContent()`** method will create an appropriate response object.

Example 47.4, "Creating a response with a `204` status" shows an example of creating a response with an **`204`** status.

Example 47.4. Creating a response with a `204` status

```
import javax.ws.rs.core.Response;

return Response.noContent().build();
```

Creating responses for redirection

The **`Response`** class provides methods for handling three of the redirection response statuses.

`303 See Other`

The **`303 See Other`** status is useful when the requested resource needs to permanently redirect the consumer to a new resource to process the request.

The **`Response`** classes **`seeOther()`** method creates a response with a **`303`** status and places the new resource URI in the message's **`Location`** field. The **`seeOther()`** method takes a single parameter that specifies the new URI as a **`java.net.URI`** object.

`304 Not Modified`

The **`304 Not Modified`** status can be used for different things depending on the nature of the request. It can be used to signify that the requested resource has not changed since a previous **`GET`** request. It can also be used to signify that a request to modify the resource did not result in the resource being changed.

The **`Response`** classes **`notModified()`** methods creates a response with a **`304`** status and sets the modified date property on the HTTP message. There are three versions of the **`notModified()`** method:

- **`notModified();`**
- **`notModified(javax.ws.rs.core.Entity tag);`**
- **`notModified(java.lang.String tag);`**

`307 Temporary Redirect`

The **`307 Temporary Redirect`** status is useful when the requested resource needs to direct the consumer to a new resource, but wants the consumer to continue using this resource to handle future requests.

The **`Response`** classes **`temporaryRedirect()`** method creates a response with a **`307`** status and places the new resource URI in the message's **`Location`** field. The **`temporaryRedirect()`** method takes a single parameter that specifies the new URI as a **`java.net.URI`** object.

Example 47.5, "Creating a response with a `304` status" shows an example of creating a response with an **304** status.

Example 47.5. Creating a response with a `304` status

```
import javax.ws.rs.core.Response;

return Response.notModified().build();
```

Creating responses to signal errors

The **Response** class provides methods to create responses for two basic processing errors:

- **`serverError()();`**—creates a response with a status of **`500 Internal Server Error`**.
- **`notAcceptable()(java.util.List<javax.ws.rs.core.Variant> variants);`**—creates a response with a **`406 Not Acceptable`** status and an entity body containing a list of acceptable resource types.

Example 47.6, "Creating a response with a `500` status" shows an example of creating a response with an **500** status.

Example 47.6. Creating a response with a `500` status

```
import javax.ws.rs.core.Response;

return Response.serverError().build();
```

47.2.3. Handling more advanced responses

Overview

The **Response** class methods provide short cuts for creating responses for common cases. When you need to address more complicated cases such as specifying cache control directives, adding custom HTTP headers, or sending a status not handled by the **Response** class, you need to use the **`ResponseBuilder`** classes methods to populate the response before using the **`build()`** method to generate the response object.

TIP

As discussed in the section called "Getting a response builder" , you can use the Apache CXF **`ResponseBuilderImpl`** class to create a response builder instance that can be manipulated directly.

Adding custom headers

Custom headers are added to a response using the **`ResponseBuilder`** class' **`header()`** method. The **`header()`** method takes two parameters:

- *name*—a string specifying the name of the header

- ***value***—a Java object containing the data stored in the header

You can set multiple headers on the message by calling the **`header()`** method repeatedly.

Example 47.7, "Adding a header to a response" shows code for adding a header to a response.

Example 47.7. Adding a header to a response

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.header("username", "joe");
Response r = builder.build();
```

Adding a cookie

Custom headers are added to a response using the **`ResponseBuilder`** class' **`cookie()`** method. The **`cookie()`** method takes one or more cookies. Each cookie is stored in a **`javax.ws.rs.core.NewCookie`** object. The easiest of the **`NewCookie`** class' contructors to use takes two parameters:

- ***name***—a string specifying the name of the cookie
- ***value***—a string specifying the value of the cookie

You can set multiple cookies by calling the **`cookie()`** method repeatedly.

Example 47.8, "Adding a cookie to a response" shows code for adding a cookie to a response.

Example 47.8. Adding a cookie to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.NewCookie;

NewCookie cookie = new NewCookie("username", "joe");

Response r = Response.ok().cookie(cookie).build();
```

WARNING

Calling the **`cookie()`** method with a **`null`** parameter list erases any cookies already associated with the response.

Setting the response status

When you want to return a status other than one of the statuses supported by the **Response** class' helper methods, you can use the **ResponseBuilder** class' **status()** method to set the response's status code. The **status()** method has two variants. One takes an int that specifies the response code. The other takes a **Response.Status** object to specify the response code.

The **Response.Status** class is an enumeration enclosed in the **Response** class. It has entries for most of the defined HTTP response codes.

Example 47.9, "Adding a header to a response" shows code for setting the response status to **404 Not Found**.

Example 47.9. Adding a header to a response

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(404);
Response r = builder.build();
```

Setting cache control directives

The **ResponseBuilder** class' **cacheControl()** method allows you to set the cache control headers on the response. The **cacheControl()** method takes a **javax.ws.rs.CacheControl** object that specifies the cache control directives for the response.

The **CacheControl** class has methods that correspond to all of the cache control directives supported by the HTTP specification. Where the directive is a simple on or off value the setter method takes a boolean value. Where the directive requires a numeric value, such as the **max-age** directive, the setter takes an int value.

Example 47.10, "Adding a header to a response" shows code for setting the **no-store** cache control directive.

Example 47.10. Adding a header to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.CacheControl;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

CacheControl cache = new CacheControl();
cache.setNoCache(true);

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.cacheControl(cache);
Response r = builder.build();
```

47.3. RETURNING ENTITIES WITH GENERIC TYPE INFORMATION

Overview

There are occasions where the application needs more control over the MIME type of the returned object or the entity provider used to serialize the response. The JAX-RS **`javax.ws.rs.core.GenericEntity<T>`** class provides finer control over the serializing of entities by providing a mechanism for specifying the generic type of the object representing the entity.

Using a `GenericEntity<T>` object

One of the criteria used for selecting the entity provider that serializes a response is the generic type of the object. The generic type of an object represents the Java type of the object. When a common Java type or a JAXB object is returned, the runtime can use Java reflection to determine the generic type. However, when a JAX-RS **`Response`** object is returned, the runtime cannot determine the generic type of the wrapped entity and the actual Java class of the object is used as the Java type.

To ensure that the entity provider is provided with correct generic type information, the entity can be wrapped in a **`GenericEntity<T>`** object before being added to the **`Response`** object being returned.

Resource methods can also directly return a **`GenericEntity<T>`** object. In practice, this approach is rarely used. The generic type information determined by reflection of an unwrapped entity and the generic type information stored for an entity wrapped in a **`GenericEntity<T>`** object are typically the same.

Creating a `GenericEntity<T>` object

There are two ways to create a **`GenericEntity<T>`** object:

1. Create a subclass of the **`GenericEntity<T>`** class using the entity being wrapped. Example 47.11, "Creating a **`GenericEntity<T>`** object using a subclass" shows how to create a **`GenericEntity<T>`** object containing an entity of type **`List<String>`** whose generic type will be available at runtime.

   Example 47.11. Creating a `GenericEntity<T>` object using a subclass

   ```
   import javax.ws.rs.core.GenericEntity;

   List<String> list = new ArrayList<String>();
   ...
   GenericEntity<List<String>> entity =
     new GenericEntity<List<String>>(list) {};
   Response response = Response.ok(entity).build();
   ```

   TIP

   The subclass used to create a **`GenericEntity<T>`** object is typically anonymous.

2. Create an instance directly by supplying the generic type information with the entity. Example 47.12, "Directly instantiating a **`GenericEntity<T>`** object" shows how to create a response containing an entity of type **`AtomicInteger`**.

   Example 47.12. Directly instantiating a `GenericEntity<T>` object

   ```
   import javax.ws.rs.core.GenericEntity;
   ```

```
AtomicInteger result = new AtomicInteger(12);
GenericEntity<AtomicInteger> entity =
  new GenericEntity<AtomicInteger>(result,

result.getClass().getGenericSuperclass());
Response response = Response.ok(entity).build();
```

CHAPTER 48. HANDLING EXCEPTIONS

Abstract

When possible, exceptions caught by a resource method should cause a useful error to be returned to the requesting consumer. JAX-RS resource methods can throw a **WebApplicaitonException** exception. You can also provide **ExceptionMapper<E>** implementations to map exceptions to appropriate responses.

48.1. USING WEBAPPLICAITONEXCEPTION EXCEPTIONS TO REPORT ERRORS

Overview

The JAX-RS API introduced the **WebApplicationException** runtime exception to provide an easy way for resource methods to create exceptions that are appropriate for RESTful clients to consume. **WebApplicationException** exceptions can include a **Response** object that defines the entity body to return to the originator of the request. It also provides a mechanism for specifying the HTTP status code to be returned to the client if no entity body is provided.

Creating a simple exception

The easiest means of creating a **WebApplicaitonException** exception is to use either the no argument constructor or the constructor that wraps the original exception in a **WebApplicationException** exception. Both constructors create a **WebApplicaitonException** with an empty response.

When an exception created by either of these constructors is thrown, the runtime returns a response with an empty entity body and a status code of **500 Server Error**.

Setting the status code returned to the client

When you want to return an error code other than **500**, you can use one of the four **WebApplicaitonException** constructors that allow you to specify the status. Two of these constructors, shown in Example 48.1, "Creating a **WebApplicationException** with a status code", take the return status as an integer.

Example 48.1. Creating a WebApplicationException with a status code

```
WebApplicationException(int status);
WebApplicationException(java.lang.Throwable cause,
                        int status);
```

The other two, shown in Example 48.2, "Creating a **WebApplicationException** with a status code" take the response status as an instance of **Response.Status**.

Example 48.2. Creating a WebApplicationException with a status code

```
WebApplicationException(javax.ws.rs.core.Response.Status status);
WebApplicationException(java.lang.Throwable cause,
                        javax.ws.rs.core.Response.Status status);
```

When an exception created by either of these constructors is thrown, the runtime returns a response with an empty entity body and the specified status code.

Providing an entity body

If you want a message to be sent along with the exception, you can use one of the **`WebApplicationException`** constructors that takes a **`Response`** object. The runtime uses the **`Response`** object to create the response sent to the client. The entity stored in the response is mapped to the entity body of the message and the status field of the response is mapped to the HTTP status of the message.

Example 48.3, "Sending a message with an exception" shows code for returning a text message to a client containing the reason for the exception and sets the HTTP message status to **`409 Conflict`**.

Example 48.3. Sending a message with an exception

```
import javax.ws.rs.core.Response;
import javax.ws.rs.WebApplicationException;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

...
ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(Response.Status.CONFLICT);
builder.entity("The requested resource is conflicted.");
Response response = builder.build();
throw WebApplicationException(response);
```

Extending the generic exception

It is possible to extend the **`WebApplicationException`** exception. This would allow you to create custom exceptions and eliminate some boiler plate code.

Example 48.4, "Extending **`WebApplicationException`**" shows a new exception that creates a similar response to the code in Example 48.3, "Sending a message with an exception" .

Example 48.4. Extending `WebApplicationException`

```
public class ConflicteddException extends WebApplicationException
{
  public ConflictedException(String message)
  {
    ResponseBuilderImpl builder = new ResponseBuilderImpl();
    builder.status(Response.Status.CONFLICT);
    builder.entity(message);
    super(builder.build());
     }
}

...
throw ConflictedException("The requested resource is conflicted.");
```

48.2. MAPPING EXCEPTIONS TO RESPONSES

Overview

There are instances where throwing a **WebApplicationException** exception is impractical or impossible. For example, you may not want to catch all possible exceptions and then create a **WebApplicationException** for them. You may also want to use custom exceptions that make working with your application code easier.

To handle these cases the JAX-RS API allows you to implement a custom exception provider that generates a **Response** object to send to a client. Custom exception providers are created by implementing the **ExceptionMapper<E>** interface. When registered with the Apache CXF runtime, the custom provider will be used whenever an exception of type E is thrown.

How exception mappers are selected

Exception mappers are used in two cases:

- When a **WebApplicationException**, or one of its subclasses, with an empty entity body is thrown, the runtime will check to see if there is an exception mapper that handles **WebApplicationException** exceptions. If there is the exception mapper is used to create the response sent to the consumer.
- When any exception other than a **WebApplicationException** exception, or one of its subclasses, is thrown, the runtime will check for an appropriate exception mapper. An exception mapper is selected if it handles the specific exception thrown. If there is not an exception mapper for the specific exception that was thrown, the exception mapper for the nearest superclass of the exception is selected.

If an exception mapper is not found for an exception, the exception is wrapped in an **ServletException** exception and passed onto the container runtime. The container runtime will then determine how to handle the exception.

Implementing an exception mapper

Exception mappers are created by implementing the **javax.ws.rs.ext.ExceptionMapper<E>** interface. As shown in Example 48.5, “Exception mapper interface” , the interface has a single method, **toResponse()**, that takes the original exception as a parameter and returns a **Response** object.

Example 48.5. Exception mapper interface

```
public interface ExceptionMapper<E extends java.lang.Throwable>
{
  public Response toResponse(E exception);
}
```

The **Response** object created by the exception mapper is processed by the runtime just like any other **Response** object. The resulting response to the consumer will contain the status, headers, and entity body encapsulated in the **Response** object.

Exception mapper implementations are considered providers by the runtime. Therefor they must be decorated with the **@Provder** annotation.

If an exception occurs while the exception mapper is building the **Response** object, the runtime will a response with a status of **500 Server Error** to the consumer.

Example 48.6, "Mapping an exception to a response" shows an exception mapper that intercepts Spring **AccessDeniedException** exceptions and generates a response with a **403 Forbidden** status and an empty entity body.

Example 48.6. Mapping an exception to a response

```
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;

import org.springframework.security.AccessDeniedException;

@Provider
public class SecurityExceptionMapper implements
ExceptionMapper<AccessDeniedException>
{

  public Response toResponse(AccessDeniedException exception)
  {
    return Response.status(Response.Status.FORBIDDEN).build();
  }

}
```

The runtime will catch any **AccessDeniedException** exceptions and create a **Response** object with no entity body and a status of **403**. The runtime will then process the **Response** object as it would for a normal response. The result is that the consumer will receive an HTTP response with a status of **403**.

Registering an exception mapper

Before a JAX-RS application can use an exception mapper, the exception mapper must be registered with the runtime. Exception mappers are registered with the runtime using the **jaxrs:providers** element in the application's configuration file.

The **jaxrs:providers** element is a child of the **jaxrs:server** element and contains a list of **bean** elements. Each **bean** element defines one exception mapper.

Example 48.7, "Registering exception mappers with the runtime" shows a JAX-RS server configured to use a an exception mapper.

Example 48.7. Registering exception mappers with the runtime

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException"
class="com.bar.providers.SecurityExceptionMapper"/>
```

```
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

CHAPTER 49. ENTITY SUPPORT

Abstract

The Apache CXF runtime supports a limited number of mappings between MIME types and Java objects out of the box. Developers can extend the mappings by implementing custom readers and writers. The custom readers and writers are registered with the runtime at start-up.

OVERVIEW

The runtime relies on JAX-RS **`MessageBodyReader`** and **`MessageBodyWriter`** implementations to serialize and de-serialize data between the HTTP messages and their Java representations. The readers and writers can restrict the MIME types they are capable of processing.

The runtime provides readers and writers for a number of common mappings. If an application requires more advanced mappings, a developer can provide custom implementations of the **`MessageBodyReader`** interface and/or the **`MessageBodyWriter`** interface. Custom readers and writers are registered with the runtime when the application is started.

NATIVELY SUPPORTED TYPES

Table 49.1, “Natively supported entity mappings” lists the entity mappings provided by Apache CXF out of the box.

Table 49.1. Natively supported entity mappings

Java Type	MIME Type
primitive types	**`text/plain`**
`java.lang.Number`	**`text/plain`**
byte[]	**`*/*`**
`java.lang.String`	**`*/*`**
`java.io.InputStream`	**`*/*`**
`java.io.Reader`	**`*/*`**
`java.io.File`	**`*/*`**
`javax.activation.DataSource`	**`*/*`**
`javax.xml.transform.Source`	**`text/xml`**, **`application/xml`**, **`application/*+xml`**
`javax.xml.bind.JAXBElement`	**`text/xml`**, **`application/xml`**, **`application/*+xml`**

Java Type	MIME Type
JAXB annotated objects	**text/xml**, **application/xml**, **application/*+xml**
javax.ws.rs.core.MultivaluedMap<String, String>	**application/x-www-form-urlencoded** [a]
javax.ws.rs.core.StreamingOutput	***/*** [b]

[a] This mapping is used for handling HTML form data.

[b] This mapping is only supported for returning data to a consumer.

CUSTOM READERS

Custom entity readers are responsible for mapping incoming HTTP requests into a Java type that a service's implementation can manipulate. They implement the **javax.ws.rs.ext.MessageBodyReader** interface.

The interface, shown in Example 49.1, "Message reader interface", has two methods that need implementing:

Example 49.1. Message reader interface

```
package javax.ws.rs.ext;

public interface MessageBodyReader<T>
{
  public boolean isReadable(java.lang.Class<?> type,
                            java.lang.reflect.Type genericType,
                            java.lang.annotation.Annotation[]
annotations,
                            javax.ws.rs.core.MediaType mediaType);

  public T readFrom(java.lang.Class<T> type,
                    java.lang.reflect.Type genericType,
                    java.lang.annotation.Annotation[] annotations,
                    javax.ws.rs.core.MediaType mediaType,
                    javax.ws.rs.core.MultivaluedMap<String, String>
httpHeaders,
                    java.io.InputStream entityStream)
  throws java.io.IOException, WebApplicationException;
}
```

isReadable()

The **isReadable()** method determines if the reader is capable of reading the data stream and creating the proper type of entity representation. If the reader can create the proper type of entity the method returns **true**.

Table 49.2, "Parameters used to determine if a reader can produce an entity" describes the `isReadable()` method's parameters.

Table 49.2. Parameters used to determine if a reader can produce an entity

Parameter	Type	Description
type	`Class<T>`	Specifies the actual Java class of the object used to store the entity.
genericType	`Type`	Specifies the Java type of the object used to store the entity. For example, if the message body is to be converted into a method parameter, the value will be the type of the method parameter as returned by the `Method.getGenericParameterTypes()` method.
annotations	`Annotation[]`	Specifies the list of annotations on the declaration of the object created to store the entity. For example if the message body is to be converted into a method parameter, this will be the annotations on that parameter returned by the `Method.getParameterAnnotations()` method.
mediaType	`MediatType`	Specifies the MIME type of the HTTP entity.

`readFrom()`

The `readFrom()` method reads the HTTP entity and coverts it into the desired Java object. If the reading is successful the method returns the created Java object containing the entity. If an error occurs when reading the input stream the method should throw an `IOException` exception. If an error occurs that requires an HTTP error response, an `WebApplicationException` with the HTTP response should be thrown.

Table 49.3, "Parameters used to read an entity" describes the `readFrom()` method's parameters.

Table 49.3. Parameters used to read an entity

Parameter	Type	Description
type	`Class<T>`	Specifies the actual Java class of the object used to store the entity.

Parameter	Type	Description
genericType	`Type`	Specifies the Java type of the object used to store the entity. For example, if the message body is to be converted into a method parameter, the value will be the type of the method parameter as returned by the `Method.getGenericParameterTypes()` method.
annotations	`Annotation[]`	Specifies the list of annotations on the declaration of the object created to store the entity. For example if the message body is to be converted into a method parameter, this will be the annotations on that parameter returned by the `Method.getParameterAnnotations()` method.
mediaType	`MediatType`	Specifies the MIME type of the HTTP entity.
httpHeaders	`MultivaluedMap<String, String>`	Specifies the HTTP message headers associated with the entity.
entityStream	`InputStream`	Specifies the input stream containing the HTTP entity.

IMPORTANT

This method should not close the input stream.

Before an `MessageBodyReader` implementation can be used as an entity reader, it must be decorated with the `javax.ws.rs.ext.Provider` annotation. The `@Provider` annotation alerts the runtime that the supplied implementation provides additional functionality. The implementation must also be registered with the runtime as described in the section called “Registering readers and writers”.

By default a custom entity provider handles all MIME types. You can limit the MIME types that a custom entity reader will handle using the `javax.ws.rs.Consumes` annotation. The `@Consumes` annotation specifies a comma separated list of MIME types that the custom entity provider reads. If an entity is not of a specified MIME type, the entity provider will not be selected as a possible reader.

Example 49.2, “XML source entity reader” shows an entity reader the consumes XML entities and stores them in a `Source` object.

Example 49.2. XML source entity reader

```
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamSource;

import org.w3c.dom.Document;
import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Consumes({"application/xml", "application/*+xml", "text/xml",
"text/html" })
public class SourceProvider implements MessageBodyReader<Object>
{
  public boolean isReadable(Class<?> type,
                            Type genericType,
                            Annotation[] annotations,
                            MediaType mt)
  {
    return Source.class.isAssignableFrom(type) ||
XMLSource.class.isAssignableFrom(type);
  }

  public Object readFrom(Class<Object> source,
                         Type genericType,
                         Annotation[] annotations,
                         MediaType mediaType,
                         MultivaluedMap<String, String> httpHeaders,
                         InputStream is)
  throws IOException
  {
    if (DOMSource.class.isAssignableFrom(source))
    {
      Document doc = null;
      DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
      DocumentBuilder builder;
      try
      {
        builder = factory.newDocumentBuilder();
        doc = builder.parse(is);
      }
      catch (Exception e)
      {
        IOException ioex = new IOException("Problem creating a Source
```

```
object");
        ioex.setStackTrace(e.getStackTrace());
        throw ioex;
       }

       return new DOMSource(doc);
    }
    else if (StreamSource.class.isAssignableFrom(source) ||
Source.class.isAssignableFrom(source))
    {
      return new StreamSource(is);
    }
    else if (XMLSource.class.isAssignableFrom(source))
    {
      return new XMLSource(is);
    }

    throw new IOException("Unrecognized source");
  }
}
```

CUSTOM WRITERS

Custom entity writers are responsible for mapping Java types into HTTP entities. They implement the **`javax.ws.rs.ext.MessageBodyWriter`** interface.

The interface, shown in Example 49.3, "Message writer interface", has three methods that need implementing:

Example 49.3. Message writer interface

```
package javax.ws.rs.ext;

public interface MessageBodyWriter<T>
{
  public boolean isWriteable(java.lang.Class<?> type,
                             java.lang.reflect.Type genericType,
                             java.lang.annotation.Annotation[]
annotations,
                             javax.ws.rs.core.MediaType mediaType);

  public long getSize(T t,
                      java.lang.Class<?> type,
                      java.lang.reflect.Type genericType,
                      java.lang.annotation.Annotation[] annotations,
                      javax.ws.rs.core.MediaType mediaType);

  public void writeTo(T t,
                      java.lang.Class<?> type,
                      java.lang.reflect.Type genericType,
                      java.lang.annotation.Annotation[] annotations,
                      javax.ws.rs.core.MediaType mediaType,
                      javax.ws.rs.core.MultivaluedMap<String, Object>
httpHeaders,
```

```
                        java.io.OutputStream entityStream)
    throws java.io.IOException, WebApplicationException;
}
```

isWriteable()

The **isWriteable()** method determines if the entity writer can map the Java type to the proper entity type. If the writer can do the mapping, the method returns **true**.

Table 49.4, "Parameters used to read an entity" describes the **isWritable()** method's parameters.

Table 49.4. Parameters used to read an entity

Parameter	Type	Description
type	**Class<T>**	Specifies the Java class of the object being written.
genericType	**Type**	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The **GenericEntity** class, described in Section 47.3, "Returning entities with generic type information", provides support for controlling this value.
annotations	**Annotation[]**	Specifies the list of annotations on the method returning the entity.
mediaType	**MediatType**	Specifies the MIME type of the HTTP entity.

getSize()

The **getSize()** method is called before the **writeTo()**. It returns the length, in bytes, of the entity being written. If a positive value is returned the value is written into the HTTP message's **Content-Length** header.

Table 49.5, "Parameters used to read an entity" describes the **getSize()** method's parameters.

Table 49.5. Parameters used to read an entity

Parameter	Type	Description
t	generic	Specifies the instance being written.

Parameter	Type	Description
type	`Class<T>`	Specifies the Java class of the object being written.
genericType	`Type`	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The `GenericEntity` class, described in Section 47.3, "Returning entities with generic type information", provides support for controlling this value.
annotations	`Annotation[]`	Specifies the list of annotations on the method returning the entity.
mediaType	`MediatType`	Specifies the MIME type of the HTTP entity.

`writeTo()`

The `writeTo()` method converts a Java object into the desired entity type and writes the entity to the output stream. If an error occurs when writing the entity to the output stream the method should throw an `IOException` exception. If an error occurs that requires an HTTP error response, an `WebApplicationException` with the HTTP response should be thrown.

Table 49.6, "Parameters used to read an entity" describes the `writeTo()` method's parameters.

Table 49.6. Parameters used to read an entity

Parameter	Type	Description
t	generic	Specifies the instance being written.
type	`Class<T>`	Specifies the Java class of the object being written.

Parameter	Type	Description
genericType	`Type`	Specifies the Java type of object to be written, obtained either by reflection of a resource method return type or via inspection of the returned instance. The `GenericEntity` class, described in Section 47.3, "Returning entities with generic type information", provides support for controlling this value.
annotations	`Annotation[]`	Specifies the list of annotations on the method returning the entity.
mediaType	`MediatType`	Specifies the MIME type of the HTTP entity.
httpHeaders	`MultivaluedMap<String, Object>`	Specifies the HTTP response headers associated with the entity.
entityStream	`OutputStream`	Specifies the output stream into which the entity is written.

Before a `MessageBodyWriter` implementation can be used as an entity writer, it must be decorated with the `javax.ws.rs.ext.Provider` annotation. The `@Provider` annotation alerts the runtime that the supplied implementation provides additional functionality. The implementation must also be registered with the runtime as described in the section called "Registering readers and writers".

By default a custom entity provider handles all MIME types. You can limit the MIME types that a custom entity writer will handle using the `javax.ws.rs.Produces` annotation. The `@Produces` annotation specifies a comma separated list of MIME types that the custom entity provider generates. If an entity is not of a specified MIME type, the entity provider will not be selected as a possible writer.

Example 49.4, "XML source entity writer" shows an entity writer that takes `Source` objects and produces XML entities.

Example 49.4. XML source entity writer

```
import java.io.IOException;
import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
```

```
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyWriter;
import javax.ws.rs.ext.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;

import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Produces({"application/xml", "application/*+xml", "text/xml" })
public class SourceProvider implements MessageBodyWriter<Source>
{

  public boolean isWriteable(Class<?> type,
                             Type genericType,
                             Annotation[] annotations,
                             MediaType mt)
  {
    return Source.class.isAssignableFrom(type);
  }

  public void writeTo(Source source,
                      Class<?> clazz,
                      Type genericType,
                      Annotation[] annotations,
                      MediaType mediatype,
                      MultivaluedMap<String, Object> httpHeaders,
                      OutputStream os)
  throws IOException
  {
    StreamResult result = new StreamResult(os);
    TransformerFactory tf = TransformerFactory.newInstance();
    try
    {
      Transformer t = tf.newTransformer();
      t.transform(source, result);
    }
    catch (TransformerException te)
    {
      te.printStackTrace();
      throw new WebApplicationException(te);
    }
  }

  public long getSize(Source source,
                      Class<?> type,
                      Type genericType,
                      Annotation[] annotations,
                      MediaType mt)
  {
```

```
    return -1;
  }
}
```

REGISTERING READERS AND WRITERS

Before a JAX-RS application can use any custom entity providers, the custom providers must be registered with the runtime. Providers are registered with the runtime using either the **jaxrs:providers** element in the application's configuration file or using the **JAXRSServerFactoryBean** class.

The **jaxrs:providers** element is a child of the **jaxrs:server** element and contains a list of **bean** elements. Each **bean** element defines one entity provider.

Example 49.5, "Registering entity providers with the runtime" show a JAX-RS server configured to use a set of custom entity providers.

Example 49.5. Registering entity providers with the runtime

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="isProvider"
class="com.bar.providers.InputStreamProvider"/>
      <bean id="longProvider" class="com.bar.providers.LongProvider"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

The **JAXRSServerFactoryBean** class is a Apache CXF extension that provides access to the configuration APIs. It has a **setProvider()** method that allows you to add instantiated entity providers to an application. Example 49.6, "Programmatically registering an entity provider" shows code for registering an entity provider programmatically.

Example 49.6. Programmatically registering an entity provider

```
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
...
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
...
SourceProvider provider = new SourceProvider();
sf.setProvider(provider);
...
```

CHAPTER 50. GETTING AND USING CONTEXT INFORMATION

Abstract

Context information includes detailed information about a resource's URI, the HTTP headers, and other details that are not readily available using the other injection annotations. Apache CXF provides special class that amalgamates the all possible context information into a single object.

50.1. INTRODUCTION TO CONTEXTS

Context annotation

You specify that context information is to be injected into a field or a resource method parameter using the `javax.ws.rs.core.Context` annotation. Annotating a field or parameter of one of the context types will instruct the runtime to inject the appropriate context information into the annotated field or parameter.

Types of contexts

Table 50.1, “Context types” lists the types of context information that can be injected and the objects that support them.

Table 50.1. Context types

Object	Context information
`UriInfo`	The full request URI
`HttpHeaders`	The HTTP message headers
`Request`	Information that can be used to determine the best representation variant or to determine if a set of preconditions have been set
`SecurityContext`	Information about the security of the requester including the authentication scheme in use, if the request channel is secure, and the user principle

Where context information can be used

Context information is available to the following parts of a JAX-RS application:

- resource classes
- resource methods
- entity providers
- exception mappers

Scope

All context information injected using the **@Context** annotation is specific to the current request. This is true in all cases including entity providers and exception mappers.

Adding contexts

The JAX-RS framework allows developers to extend the types of information that can be injected using the context mechanism. You add custom contexts by implementing a **Context<T>** object and registering it with the runtime.

50.2. WORKING WITH THE FULL REQUEST URI

The request URI contains a significant amount of information. Most of this information can be accessed using method parameters as described in Section 46.2.1, "Injecting data from a request URI" , however using parameters forces certain constraints on how the URI is processed. Using parameters to access the segments of a URI also does not provide a resource access to the full request URI.

You can provide access to the complete request URI by injecting the URI context into a resource. The URI is provided as a **UriInfo** object. The **UriInfo** interface provides functions for decomposing the URI in a number of ways. It can also provide the URI as a **UriBuilder** object that allows you to construct URIs to return to clients.

50.2.1. Injecting the URI information

Overview

When a class field or method parameter that is a **UriInfo** object is decorated with the **@Context** annotation, the URI context for the current request is injected into the **UriInfo** object.

Example

Example 50.1, "Injecting the URI context into a class field" shows a class with a field populated by injecting the URI context.

Example 50.1. Injecting the URI context into a class field

```
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo requestURI;
  ...
}
```

50.2.2. Working with the URI

Overview

One of the main advantages of using the URI context is that it provides access to the base URI of the service and the path segment of the URI for the selected resource. This information can be useful for a number of purposes such as making processing decisions based on the URI or calculating URIs to return as part of the response. For example if the base URI of the request contains a .com extension the service may decide to use US dollars and if the base URI contains a .co.uk extension is may decide to us British Pounds.

The **`UriInfo`** interface provides methods for accessing the parts of the URI:

- the base URI
- the resource path
- the full URI

Getting the Base URI

The *base URI* is the root URI on which the service is published. It does not contain any portion of the URI specified in any of the service's **`@Path`** annotations. For example if a service implementing the resource defined in Example 46.5, "Disabling URI decoding" were published to http://fusesource.org and a request was made on http://fusesource.org/montersforhire/nightstalker?12 the base URI would be http://fusesource.org.

Table 50.2, "Methods for accessing a resource's base URI" describes the methods that return the base URI.

Table 50.2. Methods for accessing a resource's base URI

Method	Desription
`URI getBaseUri();`	Returns the service's base URI as a**`URI`** object.
`UriBuilder getBaseUriBuilder();`	Returns the base URI as a **`javax.ws.rs.core.UriBuilder`** object. The **`UriBuilder`** class is useful for creating URIs for other resources implemented by the service.

Getting the path

The *path* portion of the request URI is the portion of the URI that was used to select the current resource. It does not include the base URI, but does include any URI template variable and matrix parameters included in the URI.

The value of the path depends on the resource selected. For example, the paths for the resources defined in Example 50.2, "Getting a resource's path" would be:

- **`rootPath`** — /monstersforhire/
- **`getterPath`** — /mostersforhire/nightstalker

  The **`GET`** request was made on /monstersforhire/nightstalker.
- **`putterPath`** — /mostersforhire/911

The **PUT** request was made on /monstersforhire/911.

Example 50.2. Getting a resource's path

```
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo rootUri;

  ...

  @GET
  public List<Monster> getMonsters(@Context UriInfo getUri)
  {
    String rootPath = rootUri.getPath();
    ...
  }

  @GET
  @Path("\{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @Context UriInfo getUri)
  {
    String getterPath = getUri.getPath();
    ...
  }

  @PUT
  @Path("\{id}")
  public void addMonster(@Encoded @PathParam("type") String type,
                         @Context UriInfo putUri)
  {
    String putterPath = putUri.getPath();
    ...
  }
  ...
}
```

Table 50.3, “Methods for accessing a resource's path” describes the methods that return the resource path.

Table 50.3. Methods for accessing a resource's path

Method	Desription
`String getPath();`	Returns the resource's path as a decoded URI.
`String getPath(boolean decode);`	Returns the resource's path. Specifying `false` disables URI decoding.

Method	Desription
`List<PathSegment> getPathSegments();`	Returns the decoded path as a list of `javax.ws.rs.core.PathSegment` objects. Each portion of the path, including matrix parameters, is placed into a unique entry in the list. For example the resource path box/round#tall would result in a list with three entries: `box`, `round`, and `tall`.
`List<PathSegment> getPathSegments(boolean decode);`	Returns the path as a list of `javax.ws.rs.core.PathSegment` objects. Each portion of the path, including matrix parameters, is placed into a unique entry in the list. Specifying `false` disables URI decoding. For example the resource path box#tall/round would result in a list with three entries: `box`, `tall`, and `round`.

Getting the full request URI

Table 50.4, "Methods for accessing the full request URI" describes the methods that return the full request URI. You have the option of returning the request URI or the absolute path of the resource. The difference is that the request URI includes the any query parameters appended to the URI and the absolute path does not include the query parameters.

Table 50.4. Methods for accessing the full request URI

Method	Desription
`URI getRequestUri();`	Returns the complete request URI, including query parameters and matrix parameters, as a `java.net.URI` object.
`UriBuilder getRequestUriBuilder();`	Returns the complete request URI, including query parameters and matrix parameters, as a `javax.ws.rs.UriBuilder` object. The `UriBuilder` class is useful for creating URIs for other resources implemented by the service.
`URI getAbsolutePath();`	Returns the complete request URI, including matrix parameters, as a `java.net.URI` object. The absolute path does not include query parameters.
`UriBuilder getAbsolutePathBuilder();`	Returns the complete request URI, including matrix parameters, as a `javax.ws.rs.UriBuilder` object. The absolute path does not include query parameters.

For a request made using the URI http://fusesource.org/montersforhire/nightstalker?12, the **`getRequestUri()`** methods would return http://fusesource.org/montersforhire/nightstalker?12. The **`getAbsolutePath()`** method would return http://fusesource.org/montersforhire/nightstalker.

50.2.3. Getting the value of URI template variables

Overview

As described in the section called "Setting the path", resource paths can contain variable segments that are bound to values dynamically. Often these variable path segments are used as parameters to a resource method as described in the section called "Getting data from the URI's path" . You can, however, also access them through the URI context.

Methods for getting the path parameters

The **`UriInfo`** interface provides two methods, shown in Example 50.3, "Methods for returning path parameters from the URI context", that return a list of the path parameters.

Example 50.3. Methods for returning path parameters from the URI context

```
MultivaluedMap<java.lang.String, java.lang.String> getPathParameters();
MultivaluedMap<java.lang.String,
java.lang.String> getPathParameters(boolean decode);
```

The **`getPathParameters()`** method that does not take any parameters automatically decodes the path parameters. If you want to disable URI decoding use **`getPathParameters(false)`**.

The values are stored in the map using their template identifiers as keys. For example if the URI template for the resource is /{color}/box/{note} the returned map will have two entries with the keys **`color`** and **`note`**.

Example

Example 50.4, "Extracting path parameters from the URI context" shows code for retrieving the path parameters using the URI context.

Example 50.4. Extracting path parameters from the URI context

```
import javax.ws.rs.Path;
import javax.ws.rs.Get;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.core.MultivaluedMap;

@Path("/monstersforhire/")
public class MonsterService

  @GET
  @Path("\{type}\{size}")
  public Monster getMonster(@Context UriInfo uri)
  {
    MultivaluedMap paramMap = uri.getPathParameters();
    String type = paramMap.getFirst("type");
```

```
        String size = paramMap.getFirst("size");
    }
}
```

CHAPTER 51. ANNOTATION INHERITANCE

Abstract

JAX-RS annotations can be inherited by subclasses and classes implementing annotated interfaces. The inheritance mechanism allows for subclasses and implementation classes to override the annotations inherited from its parents.

OVERVIEW

Inheritance is one of the more powerful mechanisms in Java because it allows developers to create generic objects that can then be specialized to meet particular needs. JAX-RS keeps this power by allowing the annotations used in mapping classes to resources to be inherited from super classes.

JAX-RS's annotation inheritance also extends to support for interfaces. Implementation classes inherit the JAX-RS annotations used in the interface they implement.

The JAX-RS inheritance rules do provide a mechanism for overriding inherited annotations. However, it is not possible to completely remove JAX-RS annotations from a construct that inherits them from a super class or interface.

INHERITANCE RULES

Resource classes inherit any JAX-RS annotations from the interface(s) it implements. Resource classes also inherit any JAX-RS annotations from any super classes they extend. Annotations inherited from a super class take precedence over annotations inherited from am interface.

In the code sample shown in Example 51.1, “Annotation inheritance”, the **`Kaijin`** class' **`getMonster()`** method inherits the **`@Path`**, **`@GET`**, and **`@PathParam`** annotations from the **`Kaiju`** interface.

Example 51.1. Annotation inheritance

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  public Monster getMonster(@PathParam("id") int id);
  ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{
  public Monster getMonster(int id)
  {
    ...
  }
  ...
}
```

OVERRIDING INHERITED ANNOTATIONS

Overriding inherited annotations is as easy as providing new annotations. If the subclass, or implementation class, provides any of its own JAX-RS annotations for a method then all of the JAX-RS annotations for that method are ignored.

In the code sample shown in Example 51.2, "Overriding annotation inheritance", the **`Kaijin`** class' **`getMonster()`** method does not inherit any of the annotations from the **`Kaiju`** interface. The implementation class overrides the **`@Produces`** annotation which causes all of the annotations from the interface to be ignored.

Example 51.2. Overriding annotation inheritance

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  @Produces("text/xml");
  public Monster getMonster(@PathParam("id") int id);
  ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{

  @GET
  @Path("/{id}")
  @Produces("application/octect-stream");
  public Monster getMonster(@PathParam("id") int id)
  {
    ...
  }
  ...
}
```

PART VII. DEVELOPING APACHE CXF INTERCEPTORS

Abstract

This guide describes how to write Apache CXF interceptors that can perform pre and post processing on messages.

CHAPTER 52. INTERCEPTORS IN THE APACHE CXF RUNTIME

Abstract

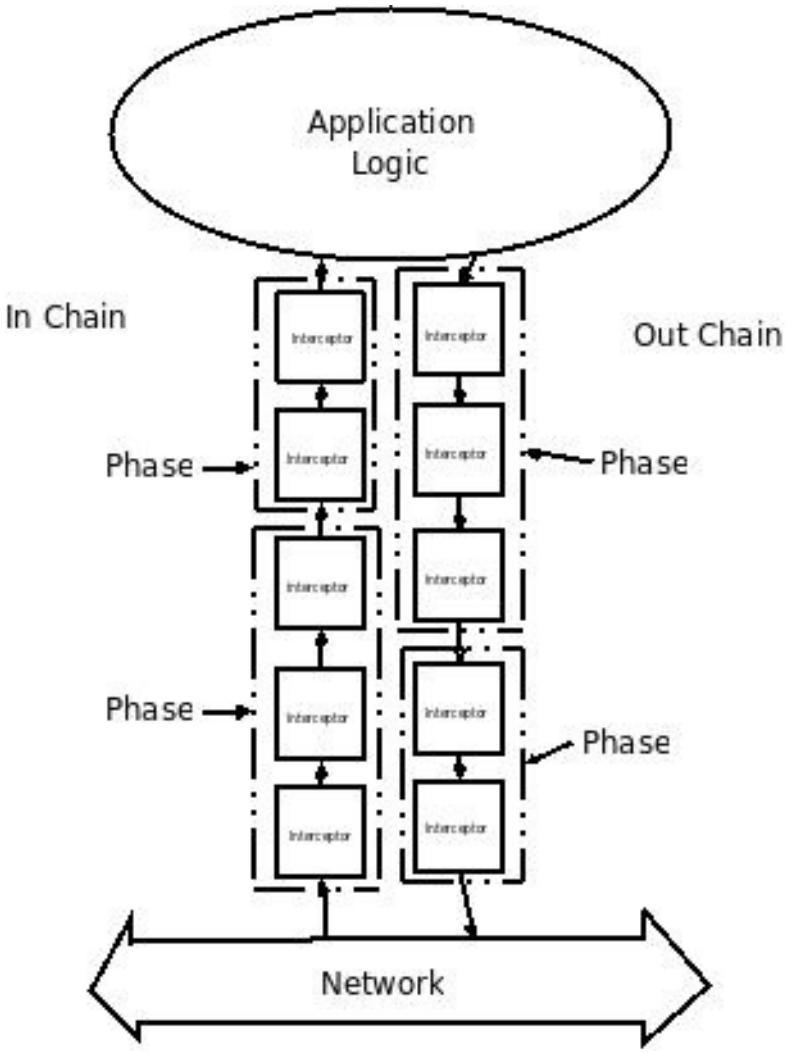
Most of the functionality in the Apache CXF runtime is implemented by interceptors. Every endpoint created by the Apache CXF runtime has three potential interceptor chains for processing messages. The interceptors in the these chains are responsible for transforming messages between the raw data transported across the wire and the Java objects handled by the endpoint's implementation code. The interceptors are organized into phases to ensure that processing happens on the proper order.

OVERVIEW

A large part of what Apache CXF does entails processing messages. When a consumer makes a invocation on a remote service the runtime needs to marshal the data into a message the service can consume and place it on the wire. The service provider must unmarshal the message, execute its business logic, and marshal the response into the appropriate message format. The consumer must then unmarshal the response message, correlate it to the proper request, and pass it back to the consumer's application code. In addition to the basic marshaling and unmarshaling, the Apache CXF runtime may do a number of other things with the message data. For example, if WS-RM is activated, the runtime must process the message chunks and acknowledgement messages before marshaling and unmarshaling the message. If security is activated, the runtime must validate the message's credentials as part of the message processing sequence.

Figure 52.1, “Apache CXF interceptor chains” shows the basic path that a request message takes when it is received by a service provider.

Figure 52.1. Apache CXF interceptor chains



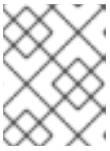
MESSAGE PROCESSING IN APACHE CXF

When a Apache CXF developed consumer invokes a remote service the following message processing sequence is started:

1. The Apache CXF runtime creates an outbound interceptor chain to process the request.
2. If the invocation starts a two-way message exchange, the runtime creates an inbound interceptor chain and a fault processing interceptor chain.
3. The request message is passed sequentially through the outbound interceptor chain.

   Each interceptor in the chain performs some processing on the message. For example, the Apache CXF supplied SOAP interceptors package the message in a SOAP envelope.
4. If any of the interceptors on the outbound chain create an error condition the chain is unwound and control is returned to the application level code.

   An interceptor chain is unwound by calling the fault processing method on all of the previously invoked interceptors.
5. The request is dispatched to the appropriate service provider.
6. When the response is received, it is passed sequentially through the inbound interceptor chain.



NOTE

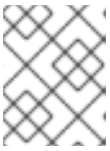
If the response is an error message, it is passed into the fault processing interceptor chain.

7. If any of the interceptors on the inbound chain create an error condition, the chain is unwound.
8. When the message reaches the end of the inbound interceptor chain, it is passed back to the application code.

When a Apache CXF developed service provider receives a request from a consumer, a similar process takes place:

1. The Apache CXF runtime creates an inbound interceptor chain to process the request message.
2. If the request is part of a two-way message exchange, the runtime also creates an outbound interceptor chain and a fault processing interceptor chain.
3. The request is passed sequentially through the inbound interceptor chain.
4. If any of the interceptors on the inbound chain create an error condition, the chain is unwound and a fault is dispatched to the consumer.

   An interceptor chain is unwound by calling the fault processing method on all of the previously invoked interceptors.
5. When the request reaches the end of the inbound interceptor chain, it is passed to the service implementation.
6. When the response is ready it is passed sequentially through the outbound interceptor chain.



NOTE

If the response is an exception, it is passed through the fault processing interceptor chain.

7. If any of the interceptors on the outbound chain create an error condition, the chain is unwound and a fault message is dispatched.
8. Once the request reaches the end of the outbound chain, it is dispatched to the consumer.

INTERCEPTORS

All of the message processing in the Apache CXF runtime is done by *interceptors*. Interceptors are POJOs that have access to the message data before it is passed to the application layer. They can do a number of things including: transforming the message, stripping headers off of the message, or validating the message data. For example, an interceptor could read the security headers off of a message, validate the credentials against an external security service, and decide if message processing can continue.

The message data available to an interceptor is determined by several factors:

- the interceptor's chain

- the interceptor's phase
- the other interceptors that occur earlier in the chain

PHASES

Interceptors are organized into *phases*. A phase is a logical grouping of interceptors with common functionality. Each phase is responsible for a specific type of message processing. For example, interceptors that process the marshaled Java objects that are passed to the application layer would all occur in the same phase.

INTERCEPTOR CHAINS

Phases are aggregated into *interceptor chains*. An interceptor chain is a list of interceptor phases that are ordered based on whether messages are inbound or outbound.

Each endpoint created using Apache CXF has three interceptor chains:

- a chain for inbound messages
- a chain for outbound messages
- a chain for error messages

Interceptor chains are primarily constructed based on the choose of binding and transport used by the endpoint. Adding other runtime features, such as security or logging, also add interceptors to the chains. Developers can also add custom interceptors to a chain using configuration.

DEVELOPING INTERCEPTORS

Developing an interceptor, regardless of its functionality, always follows the same basic procedure:

1. Determine which abstract interceptor class to extend.

   Apache CXF provides a number of abstract interceptors to make it easier to develop custom interceptors.

2. Determine the phase in which the interceptor will run.

   Interceptors require certain parts of a message to be available and require the data to be in a certain format. The contents of the message and the format of the data is partially determined by an interceptor's phase.

3. Determine if there are any other interceptors that must be executed either before or after the interceptor.

   In general, the ordering of interceptors within a phase is not important. However, in certain situations it may be important to ensure that an interceptor is executed before, or after, other interceptors in the same phase.

4. Implement the interceptor's message processing logic.

5. Implement the interceptor's fault processing logic.

   If an error occurs in the active interceptor chain after the interceptor has executed, its fault processing logic is invoked.

6. Attach the interceptor to one of the endpoint's interceptor chains.

CHAPTER 53. THE INTERCEPTOR APIS

Abstract

Interceptors implement the **PhaseInterceptor** interface which extends the base **Interceptor** interface. This interface defines a number of methods used by the Apache CXF's runtime to control interceptor execution and are not appropriate for application developers to implement. To simplify interceptor development, Apache CXF provides a number of abstract interceptor implementations that can be extended.

INTERFACES

All of the interceptors in Apache CXF implement the base **Interceptor** interface shown in Example 53.1, "Base interceptor interface".

Example 53.1. Base interceptor interface

```
package org.apache.cxf.interceptor;

public interface Interceptor<T extends Message>
{

  void handleMessage(T message) throws Fault;

  void handleFault(T message);

}
```

The **Interceptor** interface defines the two methods that a developer needs to implement for a custom interceptor:

handleMessage()

The **handleMessage()** method does most of the work in an interceptor. It is called on each interceptor in a message chain and receives the contents of the message being processed. Developers implement the message processing logic of the interceptor in this method. For detailed information about implementing the **handleMessage()** method, see Section 55.1, "Processing messages".

handleFault()

The **handleFault()** method is called on an interceptor when normal message processing has been interrupted. The runtime calls the **handleFault()** method of each invoked interceptor in reverse order as it unwinds an interceptor chain. For detailed information about implementing the **handleFault()** method, see Section 55.2, "Unwinding after an error".

Most interceptors do not directly implement the **Interceptor** interface. Instead, they implement the **PhaseInterceptor** interface shown in Example 53.2, "The phase interceptor interface". The **PhaseInterceptor** interface adds four methods that allow an interceptor the participate in interceptor chains.

Example 53.2. The phase interceptor interface

```
package org.apache.cxf.phase;
...

public interface PhaseInterceptor<T extends Message> extends
Interceptor<T>
{

    Set<String> getAfter();

    Set<String> getBefore();

    String getId();

    String getPhase();

}
```

ABSTRACT INTERCEPTOR CLASS

Instead of directly implementing the **`PhaseInterceptor`** interface, developers should extend the **`AbstractPhaseInterceptor`** class. This abstract class provides implementations for the phase management methods of the **`PhaseInterceptor`** interface. The **`AbstractPhaseInterceptor`** class also provides a default implementation of the **`handleFault()`** method.

Developers need to provide an implementation of the **`handleMessage()`** method. They can also provide a different implementation for the **`handleFault()`** method. The developer-provided implementations can manipulate the message data using the methods provided by the generic **`org.apache.cxf.message.Message`** interface.

For applications that work with SOAP messages, Apache CXF provides an **`AbstractSoapInterceptor`** class. Extending this class provides the **`handleMessage()`** method and the **`handleFault()`** method with access to the message data as an **`org.apache.cxf.binding.soap.SoapMessage`** object. **`SoapMessage`** objects have methods for retrieving the SOAP headers, the SOAP envelope, and other SOAP metadata from the message.

CHAPTER 54. DETERMINING WHEN THE INTERCEPTOR IS INVOKED

Abstract

Interceptors are organized into phases. The phase in which an interceptor runs determines what portions of the message data it can access. An interceptor can determine its location in relationship to the other interceptors in the same phase. The interceptor's phase and its location within the phase are set as part of the interceptor's constructor logic.

When developing a custom interceptor, the first thing to consider is where in the message processing chain the interceptor belongs. The developer can control an interceptor's position in the message processing chain in one of two ways:

- Specifying the interceptor's phase
- Specifying constraints on the location of the interceptor within the phase

Typically, the code specifying an interceptor's location is placed in the interceptor's constructor. This makes it possible for the runtime to instantiate the interceptor and put in the proper place in the interceptor chain without any explicit action in the application level code.

54.1. SPECIFYING AN INTERCEPTOR'S PHASE

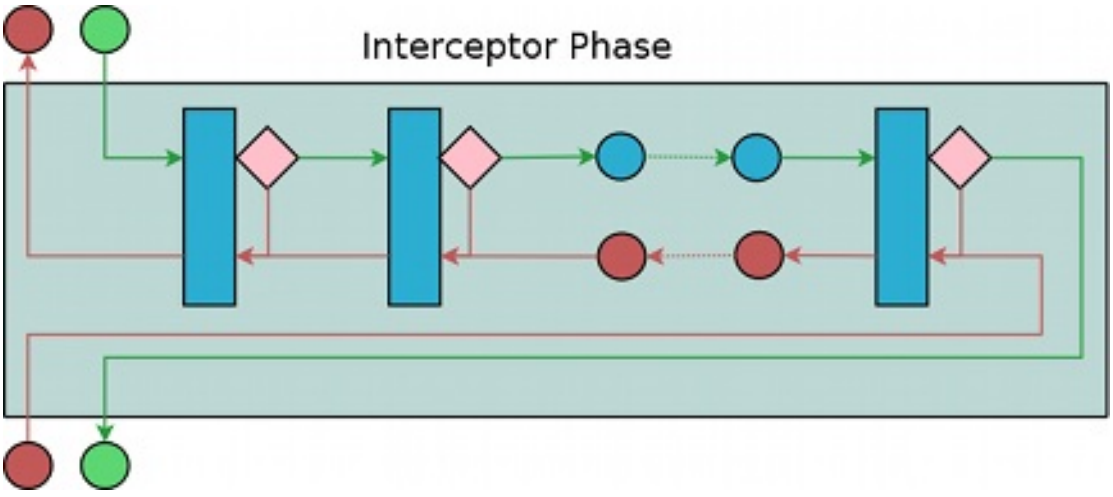
Overview

Interceptors are organized into phases. An interceptor's phase determines when in the message processing sequence it is called. Developers specify an interceptor's phase its constructor. Phases are specified using constant values provided by the framework.

Phase

Phases are a logical collection of interceptors. As shown in Figure 54.1, “An interceptor phase”, the interceptors within a phase are called sequentially.

Figure 54.1. An interceptor phase



The phases are linked together in an ordered list to form an interceptor chain and provide defined logical steps in the message processing procedure. For example, a group of interceptors in the

RECEIVE phase of an inbound interceptor chain processes transport level details using the raw message data picked up from the wire.

There is, however, no enforcement of what can be done in any of the phases. It is recommended that interceptors within a phase adhere to tasks that are in the spirit of the phase.

The complete list of phases defined by Apache CXF can be found in Appendix F, *Apache CXF Message Processing Phases*.

Specifying a phase

Apache CXF provides the `org.apache.cxf.Phase` class to use for specifying a phase. The class is a collection of constants. Each phase defined by Apache CXF has a corresponding constant in the **Phase** class. For example, the **RECEIVE** phase is specified by the value `Phase.RECEIVE`.

Setting the phase

An interceptor's phase is set in the interceptor's constructor. The `AbstractPhaseInterceptor` class defines three constructors for instantiating an interceptor:

- `public AbstractPhaseInterceptor(String phase)`—sets the phase of the interceptor to the specified phase and automatically sets the interceptor's id to the interceptor's class name.

  TIP

  This constructor will satisfy most use cases.

- `public AbstractPhaseInterceptor(String id, String phase)`—sets the interceptor's id to the string passed in as the first parameter and the interceptor's phase to the second string.

- `public AbstractPhaseInterceptor(String phase, boolean uniqueId)`—specifies if the interceptor should use a unique, system generated id. If the ***`uniqueId`*** parameter is `true`, the interceptor's id will be calculated by the system. If the ***`uniqueId`*** parameter is `false` the interceptor's id is set to the interceptor's class name.

The recommended way to set a custom interceptor's phase is to pass the phase to the `AbstractPhaseInterceptor` constructor using the `super()` method as shown in Example 54.1, "Setting an interceptor's phase".

Example 54.1. Setting an interceptor's phase

```
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
 {

    public StreamInterceptor()
   {
```

```
        super(Phase.PRE_STREAM);
    }
}
```

The **StreamInterceptor** interceptor shown in Example 54.1, "Setting an interceptor's phase" is placed into the **PRE_STREAM** phase.

54.2. CONSTRAINING AN INTERCEPTORS PLACEMENT IN A PHASE

Overview

Placing an interceptor into a phase may not provide fine enough control over its placement to ensure that the interceptor works properly. For example, if an interceptor needed to inspect the SOAP headers of a message using the SAAJ APIs, it would need to run after the interceptor that converts the message into a SAAJ object. There may also be cases where one interceptor consumes a part of the message needed by another interceptor. In these cases, a developer can supply a list of interceptors that must be executed before their interceptor. A developer can also supply a list of interceptors that must be executed after their interceptor.

IMPORTANT

The runtime can only honor these lists within the interceptor's phase. If a developer places an interceptor from an earlier phase in the list of interceptors that must execute after the current phase, the runtime will ignore the request.

Add to the chain before

One issue that arises when developing an interceptor is that the data required by the interceptor is not always present. This can occur when one interceptor in the chain consumes message data required by a later interceptor. Developers can control what a custom interceptor consumes and possibly fix the problem by modifying their interceptors. However, this is not always possible because a number of interceptors are used by Apache CXF and a developer cannot modify them.

An alternative solution is to ensure that a custom interceptor is placed before any interceptors that will consume the message data the custom interceptor requires. The easiest way to do that would be to place it in an earlier phase, but that is not always possible. For cases where an interceptor needs to be placed before one or more other interceptors the Apache CXF's **AbstractPhaseInterceptor** class provides two **addBefore()** methods.

As shown in Example 54.2, "Methods for adding an interceptor before other interceptors" , one takes a single interceptor id and the other takes a collection of interceptor ids. You can make multiple calls to continue adding interceptors to the list.

Example 54.2. Methods for adding an interceptor before other interceptors

```
public void addBefore(String i);
public void addBefore(Collection<String> i);
```

As shown in Example 54.3, "Specifying a list of interceptors that must run after the current interceptor", a developer calls the **addBefore()** method in the constuctor of a custom interceptor.

Example 54.3. Specifying a list of interceptors that must run after the current interceptor

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addBefore(HolderOutInterceptor.class.getName());
    }

...

}
```

TIP

Most interceptors use their class name for an interceptor id.

Add to the chain after

Another reason the data required by the interceptor is not present is that the data has not been placed in the message object. For example, an interceptor may want to work with the message data as a SOAP message, but it will not work if it is placed in the chain before the message is turned into a SOAP message. Developers can control what a custom interceptor consumes and possibly fix the problem by modifying their interceptors. However, this is not always possible because a number of interceptors are used by Apache CXF and a developer cannot modify them.

An alternative solution is to ensure that a custom interceptor is placed after the interceptor, or interceptors, that generate the message data the custom interceptor requires. The easiest way to do that would be to place it in a later phase, but that is not always possible. The `AbstractPhaseInterceptor` class provides two `addAfter()` methods for cases where an interceptor needs to be placed after one or more other interceptors.

As shown in Example 54.4, “Methods for adding an interceptor after other interceptors” , one method takes a single interceptor id and the other takes a collection of interceptor ids. You can make multiple calls to continue adding interceptors to the list.

Example 54.4. Methods for adding an interceptor after other interceptors

```
public void addAfter(String i);
public void addAfter(Collection<String> i);
```

As shown in Example 54.5, “Specifying a list of interceptors that must run before the current interceptor”, a developer calls the `addAfter()` method in the constuctor of a custom interceptor.

Example 54.5. Specifying a list of interceptors that must run before the current interceptor

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
```

```
            addAfter(StartingOutInterceptor.class.getName());
        }

    ...

    }
```

TIP

Most interceptors use their class name for an interceptor id.

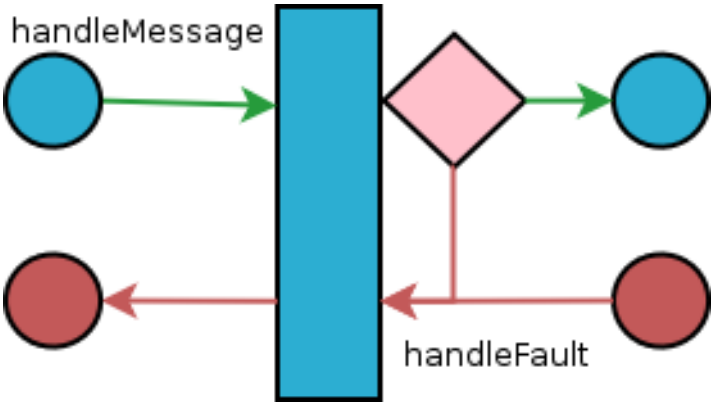
CHAPTER 55. IMPLEMENTING THE INTERCEPTORS PROCESSING LOGIC

Abstract

Interceptors are straightforward to implement. The bulk of their processing logic is in the `handleMessage()` method. This method receives the message data and manipulates it as needed. Developers may also want to add some special logic to handle fault processing cases.

Figure 55.1, "Flow through an interceptor" shows the process flow through an interceptor.

Figure 55.1. Flow through an interceptor



In normal message processing, only the `handleMessage()` method is called. The `handleMessage()` method is where the interceptor's message processing logic is placed.

If an error occurs in the `handleMessage()` method of the interceptor, or any subsequent interceptor in the interceptor chain, the `handleFault()` method is called. The `handleFault()` method is useful for cleaning up after an interceptor in the event of an error. It can also be used to alter the fault message.

55.1. PROCESSING MESSAGES

Overview

In normal message processing, an interceptor's `handleMessage()` method is invoked. It receives that message data as a `Message` object. Along with the actual contents of the message, the `Message` object may contain a number of properties related to the message or the message processing state. The exact contents of the `Message` object depends on the interceptors preceding the current interceptor in the chain.

Getting the message contents

The `Message` interface provides two methods that can be used in extracting the message contents:

- `public <T> T getContent(java.lang.Class<T> format);`
  The `getContent()` method returns the content of the message in an object of the specified class. If the contents are not available as an instance of the specified class, null is returned. The list of available content types is determined by the interceptor's location on the interceptor chain and the direction of the interceptor chain.

- **`public Collection<Attachment> getAttachments();`**
  The **`getAttachments()`** method returns a Java **`Collection`** object containing any binary attachments associated with the message. The attachments are stored in **`org.apache.cxf.message.Attachment`** objects. **`Attachment`** objects provide methods for managing the binary data.

IMPORTANT

Attachments are only available after the attachment processing interceptors have executed.

Determining the message's direction

The direction of a message can be determined by querying the message exchange. The message exchange stores the inbound message and the outbound message in separate properties.[4]

The message exchange associated with a message is retrieved using the message's **`getExchange()`** method. As shown in Example 55.1, "Getting the message exchange", **`getExchange()`** does not take any parameters and returns the message exchange as a **`org.apache.cxf.message.Exchange`** object.

Example 55.1. Getting the message exchange

```
Exchange getExchange();
```

The **Exchange** object has four methods, shown in Example 55.2, "Getting messages from a message exchange", for getting the messages associated with an exchange. Each method will either return the message as a **`org.apache.cxf.Message`** object or it will return null if the message does not exist.

Example 55.2. Getting messages from a message exchange

```
Message getInMessage();
Message getInFaultMessage();
Message getOutMessage();
Message getOutFaultMessage();
```

Example 55.3, "Checking the direction of a message chain" shows code for determining if the current message is outbound. The method gets the message exchange and checks to see if the current message is the same as the exchange's outbound message. It also checks the current message against the exchanges outbound fault message to error messages on the outbound fault interceptor chain.

Example 55.3. Checking the direction of a message chain

```
public static boolean isOutbound()
{
   Exchange exchange = message.getExchange();
        return message != null
               && exchange != null
               && (message == exchange.getOutMessage()
                   || message == exchange.getOutFaultMessage());
}
```

Example

Example 55.4, "Example message processing method" shows code for an interceptor that processes zip compressed messages. It checks the direction of the message and then performs the appropriate actions.

Example 55.4. Example message processing method

```
import java.io.IOException;
import java.io.InputStream;
import java.util.zip.GZIPInputStream;

import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
{

  ...

  public void handleMessage(Message message)
  {

        boolean isOutbound = false;
        isOutbound = message == message.getExchange().getOutMessage()
               || message ==
message.getExchange().getOutFaultMessage();

        if (!isOutbound)
        {
          try
          {
             InputStream is = message.getContent(InputStream.class);
             GZIPInputStream zipInput = new GZIPInputStream(is);
             message.setContent(InputStream.class, zipInput);
          }
          catch (IOException ioe)
          {
             ioe.printStackTrace();
           }
        }
         else
        {
         // zip the outbound message
        }
    }
  ...
}
```

55.2. UNWINDING AFTER AN ERROR

Overview

When an error occurs during the execution of an interceptor chain, the runtime stops traversing the interceptor chain and unwinds the chain by calling the **handleFault()** method of any interceptors in the chain that have already been executed.

The **handleFault()** method can be used to clean up any resources used by an interceptor during normal message processing. It can also be used to rollback any actions that should only stand if message processing completes successfully. In cases where the fault message will be passed on to an outbound fault processing interceptor chain, the **handleFault()** method can also be used to add information to the fault message.

Getting the message payload

The **handleFault()** method receives the same **Message** object as the **handleMessage()** method used in normal message processing. Getting the message contents from the **Message** object is described in the section called “Getting the message contents” .

Example

Example 55.5, “Handling an unwinding interceptor chain” shows code used to ensure that the original XML stream is placed back into the message when the interceptor chain is unwound.

Example 55.5. Handling an unwinding interceptor chain

```
@Override
public void handleFault(SoapMessage message)
{
  super.handleFault(message);
  XMLStreamWriter writer =
(XMLStreamWriter)message.get(ORIGINAL_XML_WRITER);
  if (writer != null)
  {
    message.setContent(XMLStreamWriter.class, writer);
  }
}
```

[4] It also stores inbound and outbound faults separately.

CHAPTER 56. CONFIGURING ENDPOINTS TO USE INTERCEPTORS

Abstract

Interceptors are added to an endpoint when it is included in a message exchange. The endpoint's interceptor chains are constructed from a the interceptor chains of a number of components in the Apache CXF runtime. Interceptors are specified in either the endpoint's configuration or the configuration of one of the runtime components. Interceptors can be added using either the configuration file or the interceptor API.

56.1. DECIDING WHERE TO ATTACH INTERCEPTORS

Overview

There are a number of runtime objects that host interceptor chains. These include:

- the endpoint object
- the service object
- the proxy object
- the factory object used to create the endpoint or the proxy
- the binding
- the central **Bus** object

A developer can attach their own interceptors to any of these objects. The most common objects to attach interceptors are the bus and the individual endpoints. Choosing the correct object requires understanding how these runtime objects are combined to make an endpoint.

Endpoints and proxies

Attaching interceptors to either the endpoint or the proxy is the most fine grained way to place an interceptor. Any interceptors attached directly to an endpoint or a proxy only effect the specific endpoint or proxy. This is a good place to attach interceptors that are specific to a particular incarnation of a service. For example, if a developer wants to expose one instance of a service that converts units from metric to imperial they could attach the interceptors directly to one endpoint.

Factories

Using the Spring configuration to attach interceptors to the factories used to create an endpoint or a proxy has the same effect as attaching the interceptors directly to the endpoint or proxy. However, when interceptors are attached to a factory programmatically the interceptors attached to the factory are propagated to every endpoint or proxy created by the factory.

Bindings

Attaching interceptors to the binding allows the developer to specify a set of interceptors that are applied to all endpoints that use the binding. For example, if a developer wants to force all endpoints

that use the raw XML binding to include a special ID element, they could attach the interceptor responsible for adding the element to the XML binding.

Buses

The most general place to attach interceptors is the bus. When interceptors are attached to the bus, the interceptors are propagated to all of the endpoints managed by that bus. Attaching interceptors to the bus is useful in applications that create multiple endpoints that share a similar set of interceptors.

Combining attachment points

Because an endpoint's final set of interceptor chains is an amalgamation of the interceptor chains contributed by the listed objects, several of the listed object can be combined in a single endpoint's configuration. For example, if an application spawned multiple endpoints that all required an interceptor that checked for a validation token, that interceptor would be attached to the application's bus. If one of those endpoints also required an interceptor that converted Euros into dollars, the conversion interceptor would be attached directly to the specific endpoint.

56.2. ADDING INTERCEPTORS USING CONFIGURATION

Overview

The easiest way to attach interceptors to an endpoint is using the configuration file. Each interceptor to be attached to an endpoint is configured using a standard Spring bean. The interceptor's bean can then be added to the proper interceptor chain using Apache CXF configuration elements.

Each runtime component that has an associated interceptor chain is configurable using specialized Spring elements. Each of the component's elements have a standard set of children for specifying their interceptor chains. There is one child for each interceptor chain associated with the component. The children list the beans for the interceptors to be added to the chain.

Configuration elements

Table 56.1, “Interceptor chain configuration elements” describes the four configuration elements for attaching interceptors to a runtime component.

Table 56.1. Interceptor chain configuration elements

Element	Description
`inInterceptors`	Contains a list of beans configuring interceptors to add to an endpoint's inbound interceptor chain.
`outInterceptors`	Contains a list of beans configuring interceptors to add to an endpoint's outbound interceptor chain.
`inFaultInterceptors`	Contains a list of beans configuring interceptors to add to an endpoint's inbound fault processing interceptor chain.

Element	Description
outFaultInterceptors	Contains a list of beans configuring interceptors to add to an endpoint's outbound fault processing interceptor chain.

All of the interceptor chain configuration elements take a **list** child element. The **list** element has one child for each of the interceptors being attached to the chain. Interceptors can be specified using either a **bean** element directly configuring the interceptor or a **ref** element that refers to a **bean** element that configures the interceptor.

Examples

Example 56.1, "Attaching interceptors to the bus" shows configuration for attaching interceptors to a bus' inbound interceptor chain.

Example 56.1. Attaching interceptors to the bus

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://cxf.apache.org/core"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="
       http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
       http://cxf.apache.org/transports/http/configuration
http://cxf.apache.org/schemas/configuration/http-conf.xsd
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  ...
  <bean id="GZIPStream"
class="demo.stream.interceptor.StreamInterceptor"/>

  <cxf:bus>
    <cxf:inInterceptors>
      <list>
        <ref bean="GZIPStream"/>
      </list>
    </cxf:inInterceptors>
  </cxf:bus>
</beans>
```

Example 56.2, "Attaching interceptors to a JAX-WS service provider" shows configuration for attaching an interceptor to a JAX-WS service's outbound interceptor chain.

Example 56.2. Attaching interceptors to a JAX-WS service provider

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
```

```
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <jaxws:endpoint ...>
    <jaxws:outInterceptors>
      <list>
        <bean id="GZIPStream"
class="demo.stream.interceptor.StreamInterceptor" />
      </list>
    </jaxws:outInterceptors>
  </jaxws:endpoint>
</beans>
```

More information

For more information about configuring endpoints using the Spring configuration see Part IV, “Configuring Web Service Endpoints”.

56.3. ADDING INTERCEPTORS PROGRAMMATICALLY

Interceptors can be attached to endpoints programmatically using either one of two approaches:

- the **`InterceptorProvider`** API
- Java annotations

Using the **`InterceptorProvider`** API allows the developer to attach interceptors to any of the runtime components that have interceptor chains, but it requires working with the underlying Apache CXF classes. The Java annotations can only be added to service interfaces or service implementations, but they allow developers to stay within the JAX-WS API or the JAX-RS API.

56.3.1. Using the interceptor provider API

Overview

Interceptors can be registered with any component that implements the **`InterceptorProvider`** interface shown in Example 56.3, “The interceptor provider interface” .

Example 56.3. The interceptor provider interface

```
package org.apache.cxf.interceptor;

import java.util.List;

public interface InterceptorProvider
{
    List<Interceptor<? extends Message>> getInInterceptors();

    List<Interceptor<? extends Message>> getOutInterceptors();

    List<Interceptor<? extends Message>> getInFaultInterceptors();

    List<Interceptor<? extends Message>> getOutFaultInterceptors();
```

```
}
```

The four methods in the interface allow you to retrieve each of an endpoint's interceptor chains as a Java `List` object. Using the methods offered by the Java `List` object, developers can add and remove interceptors to any of the chains.

Procedure

To use the `InterceptorProvider` API to attach an interceptor to a runtime component's interceptor chain, you must:

1. Get access to the runtime component with the chain to which the interceptor is being attached.

   Developers must use Apache CXF specific APIs to access the runtime components from standard Java application code. The runtime components are usually accessible by casting the JAX-WS or JAX-RS artifacts into the underlying Apache CXF objects.

2. Create an instance of the interceptor.

3. Use the proper get method to retrieve the desired interceptor chain.

4. Use the `List` object's `add()` method to attach the interceptor to the interceptor chain.

   TIP

   This step is usually combined with retrieving the interceptor chain.

Attaching an interceptor to a consumer

Example 56.4, "Attaching an interceptor to a consumer programmatically" shows code for attaching an interceptor to the inbound interceptor chain of a JAX-WS consumer.

Example 56.4. Attaching an interceptor to a consumer programmatically

```
package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import org.apache.cxf.endpoint.ClientProxy;
import org.apache.cxf.endpoint.ClientProxy;

public class Client
{
  public static void main(String args[])
  {
    QName serviceName = new QName("http://demo.eric.org",
"stockQuoteReporter");
1     Service s = Service.create(serviceName);
```

```
    QName portName = new QName("http://demo.eric.org",
"stockQuoteReporterPort");
    s.addPort(portName, "http://schemas.xmlsoap.org/soap/",
2 "http://localhost:9000/EricStockQuote");

3     quoteReporter proxy = s.getPort(portName, quoteReporter.class);

4     Client cxfClient = ClientProxy.getClient(proxy);

    ValidateInterceptor validInterceptor = new ValidateInterceptor();
5 6     cxfClient.getInInterceptor().add(validInterceptor);

    ...
  }
}
```

The code in Example 56.4, “Attaching an interceptor to a consumer programmatically” does the following:

1. Creates a JAX-WS **`Service`** object for the consumer.
2. Adds a port to the **`Service`** object that provides the consumer's target address.
3. Creates the proxy used to invoke methods on the service provider.
4. Gets the Apache CXF **`Client`** object associated with the proxy.
5. Creates an instance of the interceptor.
6. Attaches the interceptor to the inbound interceptor chain.

Attaching an interceptor to a service provider

Example 56.5, “Attaching an interceptor to a service provider programmatically” shows code for attaching an interceptor to a service provider's outbound interceptor chain.

Example 56.5. Attaching an interceptor to a service provider programmatically

```
package com.fusesource.demo;
import java.util.*;

import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
import org.apache.cxf.frontend.EndpointImpl;

public class stockQuoteReporter implements quoteReporter
{
  ...
  public stockQuoteReporter()
  {
1     ServerFactoryBean sfb = new ServerFactoryBean();

2       Server server = sfb.create();
          EndpointImpl endpt = server.getEndpoint();
```

```
3    AuthTokenInterceptor authInterceptor = new AuthTokenInterceptor();
4
5      endpt.getOutInterceptor().add(authInterceptor);
    }
}
```

The code in Example 56.5, “Attaching an interceptor to a service provider programmatically” does the following:

1. Creates a **`ServerFactoryBean`** object that will provide access to the underlying Apache CXF objects.
2. Gets the **`Server`** object that Apache CXF uses to represent the endpoint.
3. Gets the Apache CXF **`EndpointImpl`** object for the service provider.
4. Creates an instance of the interceptor.
5. Attaches the interceptor to the endpoint;s outbound interceptor chain.

Attaching an interceptor to a bus

Example 56.6, “Attaching an interceptor to a bus” shows code for attaching an interceptor to a bus' inbound interceptor chain.

Example 56.6. Attaching an interceptor to a bus

```
import org.apache.cxf.BusFactory;
org.apache.cxf.Bus;

...

1 Bus bus = BusFactory.getDefaultBus();

2 WatchInterceptor watchInterceptor = new WatchInterceptor();

3 bus..getInInterceptor().add(watchInterceptor);

...
```

The code in Example 56.6, “Attaching an interceptor to a bus” does the following:

1. Gets the default bus for the runtime instance.
2. Creates an instance of the interceptor.
3. Attaches the interceptor to the inbound interceptor chain.

The **`WatchInterceptor`** will be attached to the inbound interceptor chain of all endpoints created by the runtime instance.

56.3.2. Using Java annotations

Overview

Apache CXF provides four Java annotations that allow a developer to specify the interceptor chains used by an endpoint. Unlike the other means of attaching interceptors to endpoints, the annotations are attached to application-level artifacts. The artifact that is used determines the scope of the annotation's effect.

Where to place the annotations

The annotations can be placed on the following artifacts:

- the service endpoint interface(SEI) defining the endpoint

  If the annotations are placed on an SEI, all of the service providers that implement the interface and all of the consumers that use the SEI to create proxies will be affected.

- a service implementation class

  If the annotations are placed on an implementation class, all of the service providers using the implementation class will be affected.

The annotations

The annotations are all in the org.apache.cxf.interceptor package and are described in Table 56.2, “Interceptor chain annotations”.

Table 56.2. Interceptor chain annotations

Annotation	Description
`InInterceptors`	Specifies the interceptors for the inbound interceptor chain.
`OutInterceptors`	Specifies the interceptors for the outbound interceptor chain.
`InFaultInterceptors`	Specifies the interceptors for the inbound fault interceptor chain.
`OutFaultInterceptors`	Specifies the interceptors for the outbound fault interceptor chain.

Listing the interceptors

The list of interceptors is specified as a list of fully qualified class names using the syntax shown in Example 56.7, “Syntax for listing interceptors in a chain annotation” .

Example 56.7. Syntax for listing interceptors in a chain annotation

```
interceptors={"interceptor1", "interceptor2", ..., "interceptorN"}
```

Example

Example 56.8, “Attaching interceptors to a service implementation” shows annotations that attach two interceptors to the inbound interceptor chain of endpoints that use the logic provided by **SayHiImpl**.

Example 56.8. Attaching interceptors to a service implementation

```
import org.apache.cxf.interceptor.InInterceptors;

@InInterceptors(interceptors={"com.sayhi.interceptors.FirstLast",
"com.sayhi.interceptors.LogName"})
public class SayHiImpl implements SayHi
{
  ...
}
```

CHAPTER 57. MANIPULATING INTERCEPTOR CHAINS ON THE FLY

Abstract

Interceptors can reconfigure an endpoint's interceptor chain as part of its message processing logic. It can add new interceptors, remove interceptors, reorder interceptors, and even suspend the interceptor chain. Any on-the-fly manipulation is invocation-specific, so the original chain is used each time an endpoint is involved in a message exchange.

OVERVIEW

Interceptor chains only live as long as the message exchange that sparked their creation. Each message contains a reference to the interceptor chain responsible for processing it. Developers can use this reference to alter the message's interceptor chain. Because the chain is per-exchange, any changes made to a message's interceptor chain will not effect other message exchanges.

CHAIN LIFE-CYCLE

Interceptor chains and the interceptors in the chain are instantiated on a per-invocation basis. When an endpoint is invoked to participate in a message exchange, the required interceptor chains are instantiated along with instances of its interceptors. When the message exchange that caused the creation of the interceptor chain is completed, the chain and its interceptor instances are destroyed.

This means that any changes you make to the interceptor chain or to the fields of an interceptor do not persist across message exchanges. So, if an interceptor places another interceptor in the active chain only the active chain is effected. Any future message exchanges will be created from a pristine state as determined by the endpoint's configuration. It also means that a developer cannot set flags in an interceptor that will alter future message processing.

TIP

If an interceptor needs to pass information along to future instances, it can set a property in the message context. The context does persist across message exchanges.

GETTING THE INTERCEPTOR CHAIN

The first step in changing a message's interceptor chain is getting the interceptor chain. This is done using the `Message.getInterceptorChain()` method shown in Example 57.1, “Method for getting an interceptor chain”. The interceptor chain is returned as a `org.apache.cxf.interceptor.InterceptorChain` object.

Example 57.1. Method for getting an interceptor chain

```
InterceptorChain getInterceptorChain();
```

ADDING INTERCEPTORS

The **InterceptorChain** object has two methods, shown in Example 57.2, "Methods for adding interceptors to an interceptor chain", for adding interceptors to an interceptor chain. One allows you to add a single interceptor and the other allows you to add multiple interceptors.

Example 57.2. Methods for adding interceptors to an interceptor chain

```
void add(Interceptor<? extends Message> i);
void add(Collection<Interceptor<? extends Message>> i);
```

Example 57.3, "Adding an interceptor to an interceptor chain on-the-fly" shows code for adding a single interceptor to a message's interceptor chain.

Example 57.3. Adding an interceptor to an interceptor chain on-the-fly

```
void handleMessage(Message message)
{
  ...
1  AddledIntereptor addled = new AddledIntereptor();
2    InterceptorChain chain = message.getInterceptorChain();
3    chain.add(addled);
     ...
}
```

The code in Example 57.3, "Adding an interceptor to an interceptor chain on-the-fly" does the following:

1. Instantiates a copy of the interceptor to be added to the chain.

IMPORTANT

The interceptor being added to the chain should be in either the same phase as the current interceptor or a latter phase than the current interceptor.

2. Gets the interceptor chain for the current message.
3. Adds the new interceptor to the chain.

REMOVING INTERCEPTORS

The **InterceptorChain** object has one method, shown in Example 57.4, "Methods for removing interceptors from an interceptor chain", for removing an interceptor from an interceptor chain.

Example 57.4. Methods for removing interceptors from an interceptor chain

```
void remove(Interceptor<? extends Message> i);
```

Example 57.5, "Removing an interceptor from an interceptor chain on-the-fly" shows code for removing an interceptor from a message's interceptor chain.

Example 57.5. Removing an interceptor from an interceptor chain on-the-fly

```
void handleMessage(Message message)
{
  ...
  Iterator<Interceptor<? extends Message>> iterator =
            message.getInterceptorChain().iterator();
  Interceptor<?> removeInterceptor =  null;
  for (; iterator.hasNext(); ) {
    Interceptor<?> interceptor = iterator.next();
    if (interceptor.getClass().getName().equals("InterceptorClassName"))
{
      removeInterceptor = interceptor;
      break;
    }
  }

  if (removeInterceptor !=  null) {
    log.debug("Removing interceptor
{}",removeInterceptor.getClass().getName());
    message.getInterceptorChain().remove(removeInterceptor);
  }
  ...
}
```

Where ***InterceptorClassName*** is the class name of the interceptor you want to remove from the chain.

APPENDIX F. APACHE CXF MESSAGE PROCESSING PHASES

INBOUND PHASES

Table F.1, "Inbound message processing phases" lists the phases available in inbound interceptor chains.

Table F.1. Inbound message processing phases

Phase	Description
RECEIVE	Performs transport specific processing, such as determining MIME boundaries for binary attachments.
PRE_STREAM	Processes the raw data stream received by the transport.
USER_STREAM	
POST_STREAM	
READ	Determines if a request is a SOAP or XML message and builds adds the proper interceptors. SOAP message headers are also processed in this phase.
PRE_PROTOCOL	Performs protocol level processing. This includes processing of WS-* headers and processing of the SOAP message properties.
USER_PROTOCOL	
POST_PROTOCOL	
UNMARSHAL	Unmarshals the message data into the objects used by the application level code.
PRE_LOGICAL	Processes the unmarshalled message data.
USER_LOGICAL	
POST_LOGICAL	
PRE_INVOKE	
INVOKE	Passes the message to the application code. On the server side, the service implementation is invoked in this phase. On the client side, the response is handed back to the application.
POST_INVOKE	Invokes the outbound interceptor chain.

OUTBOUND PHASES

Table F.2, “Inbound message processing phases” lists the phases available in inbound interceptor chains.

Table F.2. Inbound message processing phases

Phase	Description
SETUP	Performs any set up that is required by later phases in the chain.
PRE_LOGICAL	Performs processing on the unmarshalled data passed from the application level.
USER_LOGICAL	
POST_LOGICAL	
PREPARE_SEND	Opens the connection for writing the message on the wire.
PRE_STREAM	Performs processing required to prepare the message for entry into a data stream.
PRE_PROTOCOL	Begins processing protocol specific information.
WRITE	Writes the protocol message.
PRE_MARSHAL	Marshals the message.
MARSHAL	
POST_MARSHAL	
USER_PROTOCOL	Process the protocol message.
POST_PROTOCOL	
USER_STREAM	Process the byte-level message.
POST_STREAM	
SEND	Sends the message and closes the transport stream.

IMPORTANT

Outbound interceptor chains have a mirror set of ending phases whose names are appended with **_ENDING**. The ending phases are used interceptors that require some terminal action to occur before data is written on the wire.

APPENDIX G. APACHE CXF PROVIDED INTERCEPTORS

G.1. CORE APACHE CXF INTERCEPTORS

Inbound

Table G.1, "Core inbound interceptors" lists the core inbound interceptors that are added to all Apache CXF endpoints.

Table G.1. Core inbound interceptors

Class	Phase	Description
`ServiceInvokerInterceptor`	`INVOKE`	Invokes the proper method on the service.

Outbound

The Apache CXF does not add any core interceptors to the outbound interceptor chain by default. The contents of an endpoint's outbound interceptor chain depend on the features in use.

G.2. FRONT-ENDS

JAX-WS

Table G.2, "Inbound JAX-WS interceptors" lists the interceptors added to a JAX-WS endpoint's inbound message chain.

Table G.2. Inbound JAX-WS interceptors

Class	Phase	Description
`HolderInInterceptor`	`PRE_INVOKE`	Creates holder objects for any out or in/out parameters in the message.
`WrapperClassInInterceptor`	`POST_LOGICAL`	Unwraps the parts of a wrapped doc/literal message into the appropriate array of objects.
`LogicalHandlerInInterceptor`	`PRE_PROTOCOL`	Passes message processing to the JAX-WS logical handlers used by the endpoint. When the JAX-WS handlers complete, the message is passed along to the next interceptor on the inbound chain.

Class	Phase	Description
`SOAPHandlerInterceptor`	`PRE_PROTOCOL`	Passes message processing to the JAX-WS SOAP handlers used by the endpoint. When the SOAP handlers finish with the message, the message is passed along to the next interceptor in the chain.

Table G.3, "Outbound JAX-WS interceptors" lists the interceptors added to a JAX-WS endpoint's outbound message chain.

Table G.3. Outbound JAX-WS interceptors

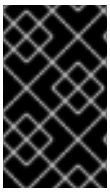
Class	Phase	Description
`HolderOutInterceptor`	`PRE_LOGICAL`	Removes the values of any out and in/out parameters from their holder objects and adds the values to the message's parameter list.
`WebFaultOutInterceptor`	`PRE_PROTOCOL`	Processes outbound fault messages.
`WrapperClassOutInterceptor`	`PRE_LOGICAL`	Makes sure that wrapped doc/literal messages and rpc/literal messages are properly wrapped before being added to the message.
`LogicalHandlerOutInterceptor`	`PRE_MARSHAL`	Passes message processing to the JAX-WS logical handlers used by the endpoint. When the JAX-WS handlers complete, the message is passed along to the next interceptor on the outbound chain.
`SOAPHandlerInterceptor`	`PRE_PROTOCOL`	Passes message processing to the JAX-WS SOAP handlers used by the endpoint. When the SOAP handlers finish processing the message, it is passed along to the next interceptor in the chain.
`MessageSenderInterceptor`	`PREPARE_SEND`	Calls back to the Destination object to have it setup the output streams, headers, etc. to prepare the outgoing transport.

JAX-RS

Table G.4, "Inbound JAX-RS interceptors" lists the interceptors added to a JAX-RS endpoint's inbound message chain.

Table G.4. Inbound JAX-RS interceptors

Class	Phase	Description
`JAXRSInInterceptor`	`PRE_STREAM`	Selects the root resource class, invokes any configured JAX-RS request filters, and determines the method to invoke on the root resource.



IMPORTANT

The inbound chain for a JAX-RS endpoint skips straight to the `ServiceInvokerInInterceptor` interceptor. No other interceptors will be invoked after the `JAXRSInInterceptor`.

Table G.5, "Outbound JAX-RS interceptors" lists the interceptors added to a JAX-RS endpoint's outbound message chain.

Table G.5. Outbound JAX-RS interceptors

Class	Phase	Description
`JAXRSOutInterceptor`	`MARSHAL`	Marshals the response into the proper format for transmission.

G.3. MESSAGE BINDINGS

SOAP

Table G.6, "Inbound SOAP interceptors" lists the interceptors added to a endpoint's inbound message chain when using the SOAP Binding.

Table G.6. Inbound SOAP interceptors

Class	Phase	Description
`CheckFaultInterceptor`	`POST_PROTOCOL`	Checks if the message is a fault message. If the message is a fault message, normal processing is aborted and fault processing is started.
`MustUnderstandInterceptor`	`PRE_PROTOCOL`	Processes the must understand headers.

Class	Phase	Description
RPCInInterceptor	**UNMARSHAL**	Unmarshals rpc/literal messages. If the message is bare, the message is passed to a **BareInInterceptor** object to deserialize the message parts.
ReadsHeadersInterceptor	**READ**	Parses the SOAP headers and stores them in the message object.
SoapActionInInterceptor	**READ**	Parses the SOAP action header and attempts to find a unique operation for the action.
SoapHeaderInterceptor	**UNMARSHAL**	Binds the SOAP headers that map to operation parameters to the appropriate objects.
AttachmentInInterceptor	**RECEIVE**	Parses the mime headers for mime boundaries, finds the *root* part and resets the input stream to it, and stores the other parts in a collection of **Attachment** objects.
DocLiteralInInterceptor	**UNMARSHAL**	Examines the first element in the SOAP body to determine the appropriate operation and calls the data binding to read in the data.
StaxInInterceptor	**POST_STREAM**	Creates an **XMLStreamReader** object from the message.
URIMappingInterceptor	**UNMARSHAL**	Handles the processing of HTTP GET methods.
SwAInInterceptor	**PRE_INVOKE**	Creates the required MIME handlers for binary SOAP attachments and adds the data to the parameter list.

Table G.7, “Outbound SOAP interceptors” lists the interceptors added to a endpoint's outbound message chain when using the SOAP Binding.

Table G.7. Outbound SOAP interceptors

Class	Phase	Description
`RPCOutInterceptor`	`MARSHAL`	Marshals rpc style messages for transmission.
`SoapHeaderOutFilterInterceptor`	`PRE_LOGICAL`	Removes all SOAP headers that are marked as inbound only.
`SoapPreProtocolOutInterceptor`	`POST_LOGICAL`	Sets up the SOAP version and the SOAP action header.
`AttachmentOutInterceptor`	`PRE_STREAM`	Sets up the attachment marshalers and the mime stuff required to process any attachments that might be in the message.
`BareOutInterceptor`	`MARSHAL`	Writes the message parts.
`StaxOutInterceptor`	`PRE_STREAM`	Creates an `XMLStreamWriter` object from the message.
`WrappedOutInterceptor`	`MARSHAL`	Wraps the outbound message parameters.
`SoapOutInterceptor`	`WRITE`	Writes the `soap:envelope` element and the elements for the header blocks in the message. Also writes an empty `soap:body` element for the remaining interceptors to populate.
`SwAOutInterceptor`	`PRE_LOGICAL`	Removes any binary data that will be packaged as a SOAP attachment and stores it for later processing.

XML

Table G.8, “Inbound XML interceptors” lists the interceptors added to a endpoint's inbound message chain when using the XML Binding.

Table G.8. Inbound XML interceptors

Class	Phase	Description

Class	Phase	Description
`AttachmentInInterceptor`	`RECEIVE`	Parses the mime headers for mime boundaries, finds the *root* part and resets the input stream to it, and then stores the other parts in a collection of `Attachment` objects.
`DocLiteralInInterceptor`	`UNMARSHAL`	Examines the first element in the message body to determine the appropriate operation and then calls the data binding to read in the data.
`StaxInInterceptor`	`POST_STREAM`	Creates an `XMLStreamReader` object from the message.
`URIMappingInterceptor`	`UNMARSHAL`	Handles the processing of HTTP GET methods.
`XMLMessageInInterceptor`	`UNMARSHAL`	Unmarshals the XML message.

Table G.9, "Outbound XML interceptors" lists the interceptors added to a endpoint's outbound message chain when using the XML Binding.

Table G.9. Outbound XML interceptors

Class	Phase	Description
`StaxOutInterceptor`	`PRE_STREAM`	Creates an `XMLStreamWriter` objects from the message.
`WrappedOutInterceptor`	`MARSHAL`	Wraps the outbound message parameters.
`XMLMessageOutInterceptor`	`MARSHAL`	Marshals the message for transmission.

CORBA

Table G.10, "Inbound CORBA interceptors" lists the interceptors added to a endpoint's inbound message chain when using the CORBA Binding.

Table G.10. Inbound CORBA interceptors

Class	Phase	Description
`CorbaStreamInInterceptor`	`PRE_STREAM`	Deserializes the CORBA message.
`BareInInterceptor`	`UNMARSHAL`	Deserializes the message parts.

Table G.11, "Outbound CORBA interceptors" lists the interceptors added to a endpoint's outbound message chain when using the CORBA Binding.

Table G.11. Outbound CORBA interceptors

Class	Phase	Description
`CorbaStreamOutInterceptor`	`PRE_STREAM`	Serializes the message.
`BareOutInterceptor`	`MARSHAL`	Writes the message parts.
`CorbaStreamOutEndingInterceptor`	`USER_STREAM`	Creates a streamable object for the message and stores it in the message context.

G.4. OTHER FEATURES

Logging

Table G.12, "Inbound logging interceptors" lists the interceptors added to a endpoint's inbound message chain to support logging.

Table G.12. Inbound logging interceptors

Class	Phase	Description
`LoggingInInterceptor`	`RECEIVE`	Writes the raw message data to the logging system.

Table G.13, "Outbound logging interceptors" lists the interceptors added to a endpoint's outbound message chain to support logging.

Table G.13. Outbound logging interceptors

Class	Phase	Description
`LoggingOutInterceptor`	`PRE_STREAM`	Writes the outbound message to the logging system.

For more information about logging see Chapter 16, *Apache CXF Logging*.

WS-Addressing

Table G.14, "Inbound WS-Addressing interceptors" lists the interceptors added to a endpoint's inbound message chain when using WS-Addressing.

Table G.14. Inbound WS-Addressing interceptors

Class	Phase	Description
MAPCodec	**PRE_PROTOCOL**	Decodes the message addressing properties.

Table G.15, "Outbound WS-Addressing interceptors" lists the interceptors added to a endpoint's outbound message chain when using WS-Addressing.

Table G.15. Outbound WS-Addressing interceptors

Class	Phase	Description
MAPAggregator	**PRE_LOGICAL**	Aggregates the message addressing properties for a message.
MAPCodec	**PRE_PROTOCOL**	Encodes the message addressing properties.

For more information about WS-Addressing see Chapter 17, *Deploying WS-Addressing*.

WS-RM

IMPORTANT

WS-RM relies on WS-Addressing so all of the WS-Addressing interceptors will also be added to the interceptor chains.

Table G.16, "Inbound WS-RM interceptors" lists the interceptors added to a endpoint's inbound message chain when using WS-RM.

Table G.16. Inbound WS-RM interceptors

Class	Phase	Description
RMInInterceptor	**PRE_LOGICAL**	Handles the aggregation of message parts and acknowledgement messages.
RMSoapInterceptor	**PRE_PROTOCOL**	Encodes and decodes the WS-RM properties from messages.

Table G.17, "Outbound WS-RM interceptors" lists the interceptors added to a endpoint's outbound message chain when using WS-RM.

Table G.17. Outbound WS-RM interceptors

Class	Phase	Description
`RMOutInterceptor`	`PRE_LOGICAL`	Handles the chunking of messages and the transmission of the chunks. Also handles the processing of acknowledgements and resend requests.
`RMSoapInterceptor`	`PRE_PROTOCOL`	Encodes and decodes the WS-RM properties from messages.

For more information about WS-RM see Chapter 18, *Enabling Reliable Messaging*.

APPENDIX H. INTERCEPTOR PROVIDERS

OVERVIEW

Interceptor providers are objects in the Apache CXF runtime that have interceptor chains attached to them. They all implement the **`org.apache.cxf.interceptor.InterceptorProvider`** interface. Developers can attach their own interceptors to any interceptor provider.

LIST OF PROVIDERS

The following objects are interceptor providers:

- **`AddressingPolicyInterceptorProvider`**
- **`ClientFactoryBean`**
- **`ClientImpl`**
- **`ClientProxyFactoryBean`**
- **`CorbaBinding`**
- **`CXFBusImpl`**
- **`org.apache.cxf.jaxws.EndpointImpl`**
- **`org.apache.cxf.endpoint.EndpointImpl`**
- **`ExtensionManagerBus`**
- **`JAXRSClientFactoryBean`**
- **`JAXRSServerFactoryBean`**
- **`JAXRSServiceImpl`**
- **`JaxWsClientEndpointImpl`**
- **`JaxWsClientFactoryBean`**
- **`JaxWsEndpointImpl`**
- **`JaxWsProxyFactoryBean`**
- **`JaxWsServerFactoryBean`**
- **`JaxwsServiceBuilder`**
- **`MTOMPolicyInterceptorProvider`**
- **`NoOpPolicyInterceptorProvider`**
- **`ObjectBinding`**
- **`RMPolicyInterceptorProvider`**

- `ServerFactoryBean`
- `ServiceImpl`
- `SimpleServiceBuilder`
- `SoapBinding`
- `WrappedEndpoint`
- `WrappedService`
- `XMLBinding`

INDEX

Symbols

B

C

D

E

F

G

H

L

M

N

O

P

S

X